SEQUENT CALCULUS:

A LOGIC AND A LANGUAGE

FOR COMPUTATION AND DUALITY

by

PAUL DOWNEN

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2017

DISSERTATION APPROVAL PAGE

Student: Paul Downen

Title: Sequent Calculus: A Logic and a Language for Computation and Duality

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Zena M. Ariola	Chairperson
Michal Young	Core Member
Boyana Norris	Core Member
Mark Lonergan	Institutional Representative

and

Scott L. Pratt	Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2017

DISSERTATION ABSTRACT

Paul Downen

Doctor of Philosophy

Department of Computer and Information Science

June 2017

Title: Sequent Calculus: A Logic and a Language for Computation and Duality

Truth and falsehood, questions and answers, construction and deconstruction; most things come in dual pairs. Duality is a mirror that reveals the new from the old via opposition. This idea appears pervasively in logic, where duality inverts “true” with “false” and “and” with “or.” However, even though programming languages are closely connected to logics, this kind of strong duality is not so apparent in practice. Sum types (disjoint tagged unions) and product types (structures) are dual concepts, but in the realm of programming, natural biases obscure their duality.

To better understand the role of duality in programming, we shift our perspective. Our approach is based on the *Curry-Howard isomorphism* which says that programs following a specification are the same as proofs for mathematical theorems. This thesis explores Gentzen’s sequent calculus, a logic steeped in duality, as a model for computational duality. By applying the Curry-Howard isomorphism to the sequent calculus, we get a language that combines dual programming concepts as equal opposites: data types found in functional languages are dual to co-data types (interface-based objects) found in object-oriented languages, control flow is dual to information flow, induction is dual to co-induction. This gives a duality-based semantics for reasoning about programs via *orthogonality*: checking safety and correctness based on a comprehensive test suite.

We use the language of the sequent calculus to apply ideas from logic to issues relevant to program compilation. The idea of *logical polarity* reveals a symmetric basis of primitive programming constructs that can faithfully represent all user-defined data and co-data types. We reflect the lessons learned back into a core language for functional languages, at the cost of symmetry, with the relationship between the sequent calculus and natural deduction. This relationship lets us derive a pure λ-calculus with user-defined data and co-data which we further extend by bringing out the implicit control-flow in functional programs. Explicit control-flow lets us share and name control the same way we share and name data, enabling a direct representation of *join points*, which are essential for tractable optimization and compilation.

CURRICULUM VITAE

NAME OF AUTHOR: Paul Downen

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
Lawrence Technological University, Southfield, MI

DEGREES AWARDED:

Doctor of Philosophy in Computer Science, 2017, University of Oregon
Bachelor of Science in Mathematics, 2010, Lawrence Technological University
Bachelor of Science in Computer Science, 2010, Lawrence Technological University
Bachelor of Science in Computer Engineering, 2010, Lawrence Technological University

AREAS OF SPECIAL INTEREST:

Programming Language Theory
Type Theory
Compilers

PROFESSIONAL EXPERIENCE:

Graduate Teaching Fellow, University of Oregon, Eugene, Oregon, September 2010 – June 2011

Graduate Research Fellow, University of Oregon, Eugene, Oregon, June 2011 – Present

Research Intern, Université Paris Diderot, INRIA, PPS, Paris, France, June 2011 – August 2011

Visiting Researcher, Université Paris Diderot, INRIA, PPS, Paris, France, November 2012 – June 2013

Visiting Researcher, Microsoft Research, Cambridge, UK, July 2015 – August 2015

GRANTS, AWARDS AND HONORS:

Oregon Doctoral Research Fellowship, University of Oregon Computer and Information Science Department, 2017

Oregon Doctoral Research Fellowship Nomination, University of Oregon Computer and Information Science Department, 2016

Upsilon Pi Epsilon Honors Society, Inducted by University of Oregon Computer and Information Science Department, 2015

Gurdeep Pall Graduate Student Fellowship, University of Oregon, 2015

Erwin & Gertrude Juilfs Scholarship in Computer and Information Science, University of Oregon, 2014

Erwin & Gertrude Juilfs Scholarship in Computer and Information Science, University of Oregon, 2012

Best GTF Award, University of Oregon Computer and Information Science Department, 2011

PUBLICATIONS:

Maurer, Luke, Downen, Paul, Ariola, Zena M., & Peyton Jones, Simon. (2017). Compiling without continuations. *Pages 482–494 of: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. PLDI '17. New York, NY, USA: ACM. Distinguished Paper Award.

Johnson-Freyd, Philip, Downen, Paul, & Ariola, Zena M. (2017). Call-by-name extensionality and confluence. *Journal of functional programming*, **27**, e12.

Downen, Paul, Maurer, Luke, Ariola, Zena M., & Peyton Jones, Simon. (2016). Sequent calculus as a compiler intermediate language. *Pages 74–88 of: Proceedings of the 21st ACM SIGPLAN international conference on functional programming*. ICFP '16. New York, NY, USA: ACM.

Johnson-Freyd, Philip, Downen, Paul, & Ariola, Zena M. (2016). First class call stacks: Exploring head reduction. *Proceedings of the workshop on continuations, WoC 2016, London, UK, April 12th 2015*. EPTCS, vol. 212.

Downen, Paul, Johnson-Freyd, Philip, & Ariola, Zena M. (2015). Structures for structural recursion. *Pages 127–139 of: Proceedings of the 20th ACM SIGPLAN international conference on functional programming*. ICFP '15. New York, NY, USA: ACM.

Downen, Paul, & Ariola, Zena M. (2014a). Compositional semantics for composable continuations: From abortive to delimited control. *Pages 109–122 of: Proceedings of the 19th ACM SIGPLAN international conference on functional programming*. ICFP '14. New York, NY, USA: ACM.

Downen, Paul, & Ariola, Zena M. (2014b). Delimited control and computational effects. *Journal of functional programming*, **24**, 1–55.

Downen, Paul, & Ariola, Zena M. (2014c). The duality of construction. *Pages 249–269 of:* Shao, Zhong (ed), *Programming languages and systems: 23rd European symposium on programming, ESOP 2014, held as part of the European joint conferences on theory and practice of software, ETAPS 2014*. Lecture Notes in Computer Science, vol. 8410. Springer Berlin Heidelberg.

Downen, Paul, Maurer, Luke, Ariola, Zena M., & Varacca, Daniele. (2014). Continuations, processes, and sharing. *Pages 69–80 of: Proceedings of the 16th international symposium on principles and practice of declarative programming*. PPDP '14. New York, NY, USA: ACM.

Ariola, Zena M., Downen, Paul, Herbelin, Hugo, Nakata, Keiko, & Saurin, Alexis. (2012). Classical call-by-need sequent calculi: The unity of semantic artifacts. *Pages 32–46 of:* Schrijvers, Tom, & Thiemann, Peter (eds), *Functional and logic programming: 11th international symposium*. Lecture Notes in Computer Science, vol. 7294. Berlin, Heidelberg: Springer Berlin Heidelberg.

Downen, Paul, & Ariola, Zena M. (2012). A systematic approach to delimited control with multiple prompts. *Pages 234–253 of:* Seidl, Helmut (ed), *Programming languages and systems: 21st European symposium on programming, ESOP 2012, held as part of the European joint conferences on theory and practice of software, ETAPS 2012*. Lecture Notes in Computer Science, vol. 7211. Springer Berlin Heidelberg. Best Paper Award Nominee.

ACKNOWLEDGEMENTS

First of all, I would like to thank those that have supported me financially during my studies. To the National Science Foundation who supported me through the grants 0917329 “A Foundation for Effects” and 1423617 “SEQUBE: A Sequent Calculus Foundation for High- Level and Intermediate Programming Languages.” And to the University of Oregon and donors Gurdeep Pall and John Juilfs who supported me through fellowships and scholarships awarded by the university and Computer and Information Science department.

I can’t thank my advisor Zena Ariola enough for the countless hours she has dedicated to mentoring me. I could not have hoped for a more attentive and encouraging advisor, and our frequent discussions and collaborations throughout my time at the University of Oregon has shaped this thesis in so many ways, both big and small. I would also like to thank other professors at the University of Oregon—Michal Young, Daniel Lowd, Boyana Norris, Hank Childs, Kathleen Freeman and others—who have given their time to advise me in matters outside research during my time here.

I would like to thank my office mates and co-members of the Oregon programming languages group, Luke Maurer and Philip Johnson-Freyd, for our collaborations and making my years at Oregon all the better. Through our joint work, they have both had their direct influence on this thesis, pulling it in different directions that I would not have gone on my own, and helped paint a fuller and brighter picture through their own work.

I am grateful for the gracious hosts that have welcomed me during my studies. To Alexis Saurin, Hugo Herbelin, and Pierre-Louis Curien at INRIA, I would like to thank them for inviting me to Paris and expanding my horizons. Much of what I learned about logic I owe to them, which has shaped this thesis. To Simon Peyton

Jones, Andrew Kennedy, and others at Microsoft Research, I would like to thank them for hosting Luke Maurer and I at Cambridge while we worked on applying our ideas to the Glasgow Haskell Compiler (GHC). Simon Peyton Jones' invaluable advice and positive attitude helped us turn the theories into real benefits for Haskell programmers. And to Iavor Diatchki and others at Galois, I would like to thank them for graciously lending their time to help kick off the work on GHC.

There have been many visitors to the University of Oregon while I was a student which enriched my experience here: Olivier Danvy, Keiko Nakata, Alexis Saurin, Pierre-Louis Curien, Marco Gaboardi, Kenichi Asai, and Jacob Johannsen. To each, I would like to say thank you for taking the time to share your ideas to me and others here at Oregon.

And finally, I would like to thank my family. To my parents Dale and Terry Downen for giving their support while I moved across the country to pursue these studies. And to my husband Chris Hoffman for his tremendous patience, support, and encouragement that made this thesis possible.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

CHAPTER I

Introduction

Truth and falsehood, questions and answers, construction and deconstruction; as Alcmaeon (510BC) once said, most things come in dual pairs. Duality is a guiding force, a mirror that reveals the new from the old via opposition. This idea appears pervasively in logic, where duality is expressed by negation that inverts "true" with "false" and "and" with "or." However, even though the theory of programming languages is closely connected to logic, this kind of strong duality is not so apparent in the practice of programming. For example, sum types (disjoint tagged unions) and pair types (structures) are related to dual concepts. But in the realm of programming, the duality between these two features is not easy to see, much less use for any practical purpose.

The situation is even worse for more complicated language features, where two concepts, both important to the theory and practice of programming, are connected by duality but one is well understood while the other is enigmatic and underdeveloped. In the case of recursion and looping, inductive data types (like lists and trees of arbitrary, but finite, size) are known to be dual to co-inductive infinite processes (like streams of input or servers that are indefinitely available) (Hagino, 1987).[1] However, while proof assistants like Coq (Coq 8.4, 2012) have a sophisticated treatment of induction, their treatment of co-induction is problematic (Giménez, 1996; Oury, 2008). The bias towards induction and inadequate treatment of co-induction in type theory and proof assistants is a road block for program verification and correctness.

Our main philosophy for approaching these questions is known as the *Curry-Howard isomorphism* or *proofs-as-programs* paradigm (Curry *et al.*, 1958; Howard, 1980; de Bruijn, 1968). The Curry-Howard isomorphism reveals a deep and profound connection between logic and programming wherein mathematical proofs *are* algorithmic programs. The canonical example of the isomorphism is the correspondence between Gentzen's (1935a) natural deduction, a system that formalizes common mathematical reasoning by laying down the rules of *intuitionistic logic*, and

[1] In general, adding the prefix "co-" to a term or concept means "the dual of that thing," and we use the shorthand "(co-)thing" to mean "both thing and co-thing."

Church's (1932) λ-calculus, one of the first models of computation and the foundation for functional programming languages. The rules for justifying proofs in intuitionistic logic correspond exactly to the rules for writing programs in functional languages, and simplifying proofs corresponds to running programs. This connection has led technical advances flow both ways: not only can we use mathematics to help write programs in functional languages, but we can also write programs to help develop mathematics with proof assistants. However, the λ-calculus is not an ideal setting for studying duality in computation. Dualities that are simple in other settings, like the De Morgan laws in logic, are far from obvious in the λ-calculus. The root of the problem is related to a lack of symmetry: natural deduction is only concerned with verifying truth and the λ-calculus is only concerned with producing results.

Natural deduction is not the only logic, however. In fact, natural deduction has a twin sibling called the *sequent calculus*, born at the same time within the seminal paper of Gentzen (1935a). Whereas the rules of natural deduction more closely mimic the reasoning that might occur in the minds of mathematicians, the rules of the sequent calculus are themselves easier to reason about, for example, if we want to show that the logic is consistent. Furthermore, unlike natural deduction's presentation of intuitionistic logic, Gentzen's sequent calculus provides a native language for *classical logic* which admits additional reasoning principles like proof by contradiction: if a logical statement cannot be false, then it must be true. As a consequence, the sequent calculus clarifies and reifies the many dualities of classical logic as pleasant symmetries baked into the very structure of its rules. In this formal system of logic, equal attention is given to falsity and truth, to assumptions and conclusions, such that there is perfect symmetry. Yet, even though these two systems look very different from each other and have their own distinct advantages and limitations, they are closely connected and give us different perspectives into the underlying phenomena of logic.

When interpreted as a programming language, the natural symmetries of the sequent calculus reveal hidden dualities in programming—input and output, production and consumption, construction and deconstruction, structure and pattern—and makes them a prominent part of the computational model. Fundamentally, the sequent calculus expresses computation as an interaction between two opposed entities: a *producer* representing a program that creates information, and a *consumer* representing an environment or context that observes information. Computation then occurs as a communication protocol allowing a producer and consumer to speak to one another. This two-party method of computation gives a different view of computation than the one shown by the λ-calculus. In particular, programs in the sequent calculus can also be seen as configurations of an abstract machine (Ariola *et al.*, 2009a), in which the evaluation context is reified as a syntactic object that may be directly manipulated. And due to the connection between classical logic (Griffin, 1990) and control operators like Scheme's (Kelsey *et al.*, 1998) callcc or Felleisen's (1992) $\mathcal{C}$, the built-in classicality of the sequent calculus also gives an effectful language for manipulating control flow.

The computational interpretation of the sequent calculus is not just an intellectual curiosity. Thanks to the relationship between natural deduction and the sequent calculus as sibling logics (Gentzen, 1935b), the sequent calculus gives us another angle for investigating real issues that arise in the λ-calculus and functional programming, from source languages down to the machine. For example, McBride (Singh *et al.*, 2011) points out how the poor foundation for the computational interpretation of co-induction is a road block for program verification and correctness, which is in contrast to the robust and powerful treatment of induction in functional languages and proof assistants. However, we show here how the symmetries of the sequent calculus show us how both induction and co-induction can be represented as equal and opposite reasoning principles under the unifying umbrella of *structural recursion* for both ordinary recursive types and generalized algebraic datatypes (*a.k.a.* GADTs). This computational symmetry between induction and co-induction is based on the duality between data types in functional languages and co-data types as objects, and gives a more robust way for proof assistants to handle recursion in infinite objects.

Moving down into the intermediate representation of programs that exists within optimizing compilers, the logic of the sequent calculus shows how compilers can use continuations in a more direct way with a "strategically defunctionalized" (Reynolds, 1998) *continuation-passing style* (CPS). This compromise between continuation-passing and direct style makes it possible to transfer techniques between CPS (Appel, 1992) and *static single assignment* (SSA) (Cytron *et al.*, 1991) compilers like SML/NJ with direct style compilers like the Glasgow Haskell Compiler. For example, CPS can faithfully represent *join points* in control flow (Kennedy, 2007), whereas direct style can use arbitrary transformations expressed in terms of the original program (Peyton Jones *et al.*, 2001). Finally, the sequent calculus can also be interpreted as an even lower-level, machine-like language for functional programs (Ohori, 1999), which can be used to reason about fine details like manual memory management (Ohori, 2003). Therefore, the computational interpretation of the sequent calculus acts like a beacon illuminating murky areas in both the design and implementation of functional languages.

Overview

The structure of this dissertation can be broken down into three major parts. First, Chapters II to IV review the background on the Curry-Howard isomorphism for logics and languages based on natural deduction and sequent calculus. Second, Chapters V and VI give the design and semantics of programming language features in the setting of the sequent calculus based on an analysis of the background in the first part. Third, Chapters VII to IX study the theory and application of the language features in the second part for the purpose of reasoning about and implementing programs.

Chapters II to VI have a linear dependency order; Chapter III depends on Chapter II, Chapter IV depends on Chapter III, and so on. After that, Chapters VII to IX depend on the preceding Chapters II to VI, but not on each other, and can be read in any order.

Chapter V is an extended and rewritten material from the previous publication (Downen & Ariola, 2014c), Chapter VI is a revised version of (Downen *et al.*, 2015), and Chapter VII uses some ideas from the supporting materials in the appendix of (Downen *et al.*, 2015).

Background

Chapter II reviews the logical system NJ of natural deduction, the core programming language represented by the simply typed λ-calculus, and the Curry-Howard correspondence between them. After considering the strength of their correspondence and its application to functional programming, the chapter concludes with some criticisms of issues in programming that are not readily addressed by these two corresponding systems.

Chapter III is about how the idea behind the Curry-Howard isomorphism leads to a foundational programming language based off the LK sequent calculus, which is an alternative view of logic from natural deduction. A core calculus—called $\mu\tilde{\mu}$—is introduced, which lies at the heart of all the languages of the sequent calculus to follow in the dissertation. The $\mu\tilde{\mu}$-calculus brings up the *fundamental dilemma* of computation in classical logic as corresponding to the need to fix an evaluation order (like eager or lazy evaluation) for programming languages. The rest of LK's logical features are layered on top of this core which lets us talk about how ideas from logic—such as de Morgan duality and *focusing*—translate to important concepts in programming.

Chapter IV is about the application of *polarity* from logic to programming. In logic, polarity tells us that types have one of two fundamental orientations—positive or negative—which can be observed from the nature of their rules and impact their meaning both in proof theory and computation. This brings into focus the connection between pattern matching (from functional programming languages) and extensionality (i.e. the idea that the only thing that can be observed about objects is how they react to stimuli), and tells us how to combine both call-by-value and call-by-name evaluation orders within a single program.

Language design

Chapter V presents a general framework that captures the previous interpretations of the sequent calculus as a programming language (from Chapters III and IV), and separates several independent concepts that were previously entangled. The main ideas of this chapter are:

- All the individual logical connectives considered previously in the dissertation can be represented by either *data* or *co-data* which are dual programming constructs to one another and represent the mechanisms that both functional and object-oriented languages use to let programmers declare new custom types.

- The impact of evaluation strategies on the behavior of programs can be described by a discipline on substitution (i.e. what could a variable in a program possibly stand for?), which lets us abstract away the differences caused by evaluation orders out of the syntactic semantics of programming languages. This abstract view of evaluation strategies encompasses the simple and canonical strategies, namely call-by-value and call-by-name, as well as more complex and nuanced strategies like call-by-need or radically non-deterministic evaluation.

- Programs can make use of multiple evaluation strategies by combining many substitution disciplines (from the previous point) which are kept separate by specifying a particular evaluation strategy for each type, so there are several distinct kinds of types with each kind corresponding to a specific strategy. This corresponds to the way that the Glasgow Haskell Compiler uses *unboxed types* (Peyton Jones & Launchbury, 1991) to distinguish the different evaluation orders of (necessarily strict) machine numbers and arrays from the otherwise lazy Haskell programs.

Chapter VI extends Chapter V with well-founded induction and co-induction, giving a fair treatment of co-induction by representing both as just specific use-cases of *structural recursion* that can't loop forever. The main ideas of this chapter are:

- Type abstraction (i.e. generics and modules) can be achieved by generalizing the language of types with type functions, and letting (co-)data types quantify over private type parameters that are not externally visible in their interface.

- Recursion in types (i.e. recursive types like lists or trees) can be achieved by recursive data or co-data declarations using both the *primitive recursion* and *noetherian recursion* principles from mathematics, where the recursive argument is an index that tracks the "size" of the type (like the length of the list or height of the tree).

- Recursion in programs (i.e. loops which must terminate on all inputs) can be achieved by abstracting over the size index to recursive types, so that the program cannot loop forever since the statically-known size must always decrease each cycle.

Theory and application

Chapter VII develops a semantics for the programming language designed in Chapters V and VI based on the idea of *orthogonality* (also known as bi-orthogonality, $\top\top$-closure, or classical realizability). This gives a model connecting compile-time types to run-time behavior useful for confirming language-wide safety properties in the style of exhaustive testing: the collection of safe programs of a type are selected from a pool of potential candidate implementations by checking them against a test suite of observations; or dually a collection of safe observations of a type are selected from the possible ones that might be considered by checking them against the blessed specification programs. The chapter begins with a general introduction to orthogonality and a comparison to negation in intuitionistic logic, and then builds a specific model for the sequent-based language which is parameterized by both the declared (co-)data types and the evaluation strategy(ies) used to interpret programs. The adequacy of the model—that is, the fact that the syntactic typing rules implies their semantic equivalent—is then applied to confirm several safety properties of the language, including type safety, strong normalization, and the soundness of (typed) extensionality laws with respect to the (untyped) operational semantics.

Chapter VIII applies the ideas from Chapter V to the problem of how polarity (from Chapter IV) informs us of a small, finite collection of data and co-data types which are capable of faithfully encoding every other (simple) type that a programmer could possibly come up with. The emphasis here is on the "faithfulness" of encodings which requires that some care is taken about which evaluation strategy is used at each point in the program, so that the encodings don't accidentally introduce the possibility of rogue behavior that the programmer's original type disallowed. To that point, this chapters give a formal verification based on a theory of type isomorphisms of the common folklore from polarized logic that complex types from both call-by-value and call-by-name functional programming languages can be represented with the primitive polarized types by sprinkling the special polarity *shift* connectives in the appropriate places. However, the broader view of evaluation strategies and (co-)data types taken here lets us consider how to encode types from call-by-need languages as well, which uses *four* (rather than just the normal two) different shifts to and from the canonical call-by-value and call-by-name strategies.

Chapter IX goes full circle, and relates back to natural deduction and the λ-calculus, demonstrating how languages from Chapters V and VI based on the sequent calculus can impact functional programming. The canonical relationship between natural deduction and the sequent calculus gives a strong, bi-directional correspondence to the intuitionistic restriction of the $\mu\tilde{\mu}$-calculus and λ-calculus family of languages. This correspondence can be applied to functional programming languages, which are based on the λ-calculus, in one of two ways: (1) in the one direction, functional programs can be compiled down to a machine-like representation based on the sequent calculus, and (2) in the other direction, theories and ideas from the sequent calculus can be translated back to the λ-calculus and the functional paradigm. Afterward, the intuitionistic restriction is lifted, and the correspondence is generalized to cover the full classical $\mu\tilde{\mu}$-calculus by generalizing the λ-calculus with first-class control. This generalization gives us a foundational language and a starting point for talking about *join points*—a general technique for efficiently representing shared control flow in programs—in direct style.

CHAPTER II

Natural Deduction

The foundations of mathematics and computation have connections that took root in the early 1900s, when Hilbert posed the *decision problem*:

> Is there an effectively calculable procedure that can decide whether a logical statement is true or false?

This question, and its negative answer, prompted an investigation into the rigorous meaning of what is "effectively computable" from Church (1936), Turing (1936), and Gödel (1934). Later on, a much deeper connection between models of computation and formalizations of logic was independently discovered and rediscovered many times (Curry *et al.*, 1958; Howard, 1980; de Bruijn, 1968). The most typical form of this amazing coincidence, now known as the *Curry-Howard isomorphism* or the *proofs-as-programs paradigm*, gives a structural isomorphism between Church's (1932) λ-calculus, a system for computing with functions, and Gentzen's (1935a) natural deduction, a system for formalizing mathematical logic. To illustrate the connection between logic and programming, we will review the two systems and show how both they reveal similar core concepts in different ways. In particular, two principles important for characterizing the meaning of various structures, which we call β and η from the tradition of the λ-calculus, arise independently in both fields of study.

Gentzen's NJ

In 1935, Gentzen formalized an intuitive model of logical reasoning called *natural deduction*, as it aimed to symbolically model the "natural" way that mathematicians reason about proofs. A proof in natural deduction is a tree-like structure made up of several *inferences*:

$$\frac{\begin{array}{c}\vdots\\ H_1\end{array} \quad \begin{array}{c}\vdots\\ H_2\end{array} \quad \dots \quad \begin{array}{c}\vdots\\ H_n\end{array}}{J}$$

where we infer the *conclusion* J from proofs of the *premises* $H_1, H_2, \dots, H_3$. The conclusion J and premises H_i are all *judgments* that make a statement about logical *propositions* (which we denote by the variables $A, B, C, \dots$) that may be true or false, such as “0 is greater than 1.” For example, we can make the basic judgment that a proposition A is true, which we will write as $\vdash A$. Proof trees are built by stacking together compatible inferences of the above form; we say that a proof tree is *closed* if all leaves of the tree end with an *axiom*—that is, the special case of an inference with zero premises—otherwise it is *open*. Open proof trees represent (partial) proofs that rely on unsubstantiated assumptions, whereas closed proof trees represent self-contained (complete) proofs.

Syntax and rules

The propositions that we deal with in logics like natural deduction are meant to represent falsifiable or verifiable claims in a particular domain of study, such as “1 + 1 = 2.” However, in their simplest form, these systems don’t account for domain-specific knowledge and leave such basic propositions as *atoms* or uninterpreted variables. Instead, the primary interest of the logic is to characterize the meaning of *connectives* that combine (zero or more) existing propositions, which are the logical glue for putting together the basic building blocks. These connectives become the central focus in Gentzen’s NJ, whose syntax and rules are given in Figure 2.1.

For example, the idea of logical conjunction is expressed formally as a connective, written $A \wedge B$ in NJ and read “A and B,” along with some associated rules of inference for building proofs involving conjunction. On the one hand, in order to deduce that $A \wedge B$ is true we may use the *introduction rule* $\wedge I$:

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B} \wedge I$$

That is to say, if we have a proof that A is true and a proof that B is true, then we have a proof that $A \wedge B$ is true. On the other hand, in order to use the fact that $A \wedge B$ is true we may use either one of the *elimination rules* $\wedge E_1$ or $\wedge E_2$:

$$\frac{\vdash A \wedge B}{\vdash A} \wedge E_1 \qquad\qquad \frac{\vdash A \wedge B}{\vdash B} \wedge E_2$$

That is to say, if we have a proof that $A \wedge B$ is true, then it must be the case that A is true and also that B is true.

$$\begin{aligned} X, Y, Z \in \mathit{PropVariable} &::= \dots \\ A, B, C \in \mathit{Proposition} &::= X \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid A \supset B \mid \forall X.A \mid \exists X.A \\ H, J \in \mathit{Judgement} &::= \vdash A \end{aligned}$$

$$\frac{}{\vdash \top}\,\top I \qquad \text{no } \top E \text{ rule} \qquad \text{no } \bot I \text{ rule} \qquad \frac{\vdash \bot}{\vdash C}\,\bot E$$

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B}\,\wedge I \qquad \frac{\vdash A \wedge B}{\vdash A}\,\wedge E_1 \qquad \frac{\vdash A \wedge B}{\vdash B}\,\wedge E_2$$

$$\frac{\vdash A}{\vdash A \vee B}\,\vee I_1 \qquad \frac{\vdash B}{\vdash A \vee B}\,\vee I_2 \qquad \frac{\vdash A \vee B \quad \begin{array}{c}\overline{\vdash A}^{\,x} \\ \vdots \\ \vdash C\end{array} \quad \begin{array}{c}\overline{\vdash B}^{\,y} \\ \vdots \\ \vdash C\end{array}}{\vdash C}\,\vee E_{x,y}$$

$$\frac{\begin{array}{c}\overline{\vdash A}^{\,x} \\ \vdots \\ \vdash B\end{array}}{\vdash A \supset B}\,\supset I_x \qquad \frac{\vdash A \supset B \quad \vdash A}{\vdash B}\,\supset E$$

$$\frac{\begin{array}{c}\vdots\ (X \notin FV(*)) \\ \vdash A\end{array}}{\vdash \forall X.A}\,\forall I_X \qquad \frac{\vdash \forall X.A}{\vdash A\{B/X\}}\,\forall E$$

$$\frac{\vdash A\{B/X\}}{\vdash \exists X.A}\,\exists I \qquad \frac{\vdash \exists X.A \quad \begin{array}{c}\overline{\vdash A}^{\,x} \\ \vdots\ (X \notin FV(*)) \\ \vdash C\end{array} \qquad (X \notin FV(C))}{\vdash C}\,\exists E_{X,x}$$

FIGURE 2.1. The NJ natural deduction system for second-order propositional logic: with truth ($\top$), falsehood ($\bot$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$), and both universal ($\forall$) and existential ($\exists$) propositional quantification.

NJ also gives an account of logical implication as a connective in natural deduction, written $A \supset B$ and read “A implies B” or “if A then B,” in a similar fashion. In order to deduce that $A \supset B$ is true we may use the introduction rule ${\supset}I$ for implication:

$$\begin{array}{c} \overline{\vdash A}\ x \\ \vdots \\ \dfrac{\vdash B}{\vdash A \supset B}\ {\supset}I_x \end{array}$$

Notice that the introduction rule for implication has a more complex form of introduction rule than the one for conjunction. In particular, the single premise of the ${\supset}I$ rule introduces a local assumption that is only visible in the proof tree of that premise. This premise says that if we can prove that B is true by assuming that A is true, then we can conclude that $A \supset B$ is true without the extra free assumption about A. As a matter of bookkeeping, the identifier x used to mark the local axiom whose scope within the overall proof is delimited by a corresponding ${\supset}I_x$ introduction rule for proving the truth of an implication. Note that this local axiom x may be used as many times as necessary in the sub-proof—be it zero times or several times—so long as it is not used outside the scope created by the ${\supset}I_x$ rule. Once we have a proof of $A \supset B$, we may make use of it with the elimination rule ${\supset}E$ for implication:

$$\dfrac{\vdash A \supset B \quad \vdash A}{\vdash B}\ {\supset}E$$

This is a formulation of the traditional reasoning principle *modus ponens*: if we believe that A implies B is true and that A is true as well, then we must believe B is true.

The last binary connective in NJ, written $A \vee B$ and read “A or B,” formalizes logical disjunction. There are two different ways to prove that $A \vee B$ is true, which corresponds to two different introduction rules ${\vee}I$ for disjunction:

$$\dfrac{\vdash A}{\vdash A \vee B}\ {\vee}I_1 \qquad\qquad \dfrac{\vdash B}{\vdash A \vee B}\ {\vee}I_2$$

If we have a proof that A is true or a proof that B is true, then we have a proof that $A \vee B$ is true. Notice how the elimination rules for $\vee$ are like upside-down versions of the introduction rules for $\wedge$. Unfortunately, making use of a proof that $A \vee B$ is true is awkward in natural deduction, compared to connectives like conjunction and implication. The elimination rule ${\vee}E$ for disjunction is the most complex one of the binary connectives of NJ:

$$\dfrac{\vdash A \lor B \quad \begin{array}{c}\overline{\vdash A}^{\;x}\\ \vdots \\ \vdash C\end{array} \quad \begin{array}{c}\overline{\vdash B}^{\;y}\\ \vdots \\ \vdash C\end{array}}{\vdash C}\lor E_{x,y}$$

This elimination rule assumes three premises: that $A \lor B$ is true, that assuming A is true lets us prove that C is true, and that assuming B is true lets us prove that C is true. The conclusion of the rule asserts that C must be true because we know how to prove it in either possible case where A or B is true. Note that the $\lor E$ elimination rule relies (twice) on the same mechanism of local assumptions for the two sub-proofs of C that was also used in the $\supset I$ introduction rule. Hence, we use the same bookkeeping identifiers connecting both local axioms x and y with the rule $\lor E_{x,y}$ that delimits their scope in the overall proof.

In the degenerate case, connectives that join zero propositions together serve as logical constants. For example, consider a connective that internalizes the notion of truth or validity into the system, written $\top$ and pronounced “true.” By its intuitive meaning, we may always deduce that $\top$ is true with no additional premise, as described by the introduction rule $\top I$:

$$\dfrac{}{\vdash \top}\top I$$

However, we can do nothing interesting with a proof that $\top$ is true. In other words, “nothing in, nothing out.” Notice how $\top$ can be understood as the nullary version of the binary connective $\land$ for conjunction: $\top$ has a single introduction rule with zero premises similar to $\land$’s two-premise introduction rule, and $\top$ has no elimination rules compared with $\land$’s two eliminations.

We can also consider a connective for internalizing the notion of falsehood, written $\bot$ and pronounced “false.” In contrast to $\top$, we should never be able to prove that $\bot$ is true in any sensible context since that would be, well, false. In other words, there is no valid introduction rule $\bot I$. But if we are in some context where $\bot$ is true for some reason, then for all intents and purposes any proposition C might as well be true, as described by the elimination rule $\bot E$:

$$\dfrac{\vdash \bot}{\vdash C}\bot E$$

Again, notice how $\bot$ can be understood as the nullary version of the binary connective $\vee$ for disjunction: $\bot$ has no introduction rules compared to $\vee$'s two introductions, and $\bot$ has a single elimination rule with $\vdash \bot$ as the only premise compared to $\vee$'s elimination rule that assumes two premises in addition to $\vdash A \vee B$.

Using connectives described above, we can also define a derived connective for negation, written $\neg A$ and pronounced "not A," which can be used to (indirectly) state that a proposition is *not* true. For example, we should intuitively expect to be able to prove $\vdash \neg\bot$ ("false is not true") in NJ but be unable to derive $\vdash \neg\top$ ("true is not true"). In lieu of treating $\neg$ as a proper connective,[1] it can be defined in terms of implication ($\supset$) and falsehood ($\bot$)

$$\neg A \triangleq A \supset \bot$$

so that the derived rules for negation that come from this encoding are:

$$\dfrac{\begin{array}{c}\overline{\vdash A}^{\,x}\\ \vdots\\ \vdash \bot\end{array}}{\vdash \neg A}\supset I_x \qquad\qquad \dfrac{\vdash \neg A \quad \vdash A}{\vdash \bot}\supset E$$

Finally, the most complex form of propositions in NJ are the *quantifiers*: logical connectives which abstract over a proposition variable (denoted by X, Y, Z, and of which there are countably many) that occurs inside of a proposition.[2] The first such quantifier is the universal quantifier, written $\forall X.A$ and pronounced "for all X, A," which codifies when the quantified proposition variable X may stand for *any* proposition. For example, NJ has the property that for any proposition A, $\vdash A \supset A$ is provable. This fact can be represented more formally by proving $\vdash \forall X.X \supset X$, where X is the universally quantified proposition variable. The second quantifier is the existential quantifier, written $\exists X.A$ and pronounced "there exists an X such that A," which codifies when the quantified proposition variable X stands for a *specific* but *unknown* proposition. For example, there are propositions in NJ that are provably

[1] Although Gentzen (1935a) did originally treat negation as a proper connective in NJ, it was defined in terms of the $\bot$ connective so that the associated introduction and elimination rules for negation are identical to the ones given here.

[2] For simplicity, we limit the presentation of NJ to second-order propositional logic. That is to say, the quantifiers $\forall$ and $\exists$ abstract over propositions themselves, as opposed to objects of some particular domain of interest like numbers.

true (such as the aforementioned $A \supset A$ or simply the trivial truth $\top$), which can be represented formally by proving $\vdash \exists X.X$.

Since both of these quantifiers bind variables in propositions, all the usual subtleties in programming languages involving static variables applies. In summary, an occurrence of a proposition variable X in a proposition A is *bound* if it is within the context of an $\forall X$ or an $\exists X$ and *free* otherwise (FV denotes the function that computes the set of free variables of a proposition), and $A\{B/X\}$ denotes the usual *capture-avoiding substitution* operation where all free occurrences of X in A are replaced with B such that all free occurrences of variables within B are still free after substitution. We also do not distinguish propositions based on the choice of bound variable names, commonly known as α *equivalence*, as stated by the two equalities for quantifiers:

$$\forall X.A\{X/Z\} =_\alpha \forall Y.A\{Y/Z\} \qquad\qquad \exists X.A\{X/Z\} =_\alpha \exists Y.A\{Y/Z\}$$

where X and Y must not be free in A. The important property α equivalence and capture-avoiding substitution is that they commute with one another, so that renaming bound variables does not affect the result of substitution up to α equivalence. Stated more formally, for all propositions A, B, and C, if $A =_\alpha B$ then $A\{C/X\} =_\alpha B\{C/X\}$. A more thorough introduction to static variables and substitution is given by Barendregt (1985) and Pierce (2002). In general, throughout this thesis we will take α-equivalence for granted whenever static variable binders are present without belaboring the formalities.

Establishing universal truths is a delicate matter, and requires the proper discipline when crafting well-formed proofs. This subtlety rears its head in the universal introduction rule $\forall I$ for proving $\forall X.A$, which requires a new form of constraint on its premise:

$$\dfrac{\begin{array}{c}\vdots\ (X \notin FV(*))\\ \vdash A\end{array}}{\vdash \forall X.A}\ \forall I_Y$$

The side condition $X \notin FV(*)$ on the proof in the premise of the $\forall I$ rule means that the variable X cannot appear free in any of the propositions in the open leaves of the sub-proof tree. Intuitively, this side condition on the variable X ensures that X is totally generic in the sub-proof, so that we do not accidentally assume anything about X that could leak into another part of the overall proof. Therefore, the $\forall I$ rule can be understood as stating that if we prove A is true where X is generic, then $\forall X.A$ must also be true. In contrast, the universal elimination rule $\forall E$ has no such side condition and can apply to any premise:

$$\dfrac{\vdash \forall X.A}{\vdash A\{B/X\}}\ \forall E$$

In other words, from a proof that $\forall X.A$ is true, any instance of A with an arbitrary B substituted for X is also true.

In contrast to the universal quantifier, establishing existential truths is easy. We may deduce that $\exists X.A$ is true by using the introduction rule $\exists I$:

$$\dfrac{\vdash A\{B/X\}}{\vdash \exists X.A}\ \exists I$$

which says that if A is true for some choice of B substituted for X, then it must be that $\exists X.A$ is true. Notice that the introduction rule for $\exists$ is like an upside-down version of the elimination rule for $\forall$; neither of the two rules impose any special criteria on their premise. However, it is harder to use the fact that $\exists X.A$ is true with the corresponding elimination rule $\exists E$:

$$\dfrac{\vdash \exists X.A \quad \begin{array}{l}\overline{\vdash A}^{\;x}\\ \ \vdots\ (X \notin FV(*))\\ \vdash C\end{array} \qquad\qquad (X \notin FV(C))}{\vdash C}\ \exists E_{X,x}$$

The same side condition $X \notin FV(*)$ that appeared in the premise of $\forall I$ also appears in the second premise of $\exists E$, so that X cannot appear free in any open leaves (besides uses of the axiom x) of the sub-proof, but additionally the existential elimination rule must also ensure that X is not free in the conclusion C. Intuitively, both of these side conditions ensure that both the result $\vdash C$ as well as its sub-proof is generic in the choice of X. Therefore, the $\exists E$ rule can be understood as stating that if we can prove that $\exists X.A$ is true and that C can be proved true from assuming A is true with a generic X, then C must be true in general.

Example 2.1. Consider how we might build a proof that $((A \wedge B) \wedge C) \supset (B \wedge A)$ is true. To start searching for a proof, we may begin with our goal $\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)$ at the bottom of the proof tree, and then try to simplify the goal by applying the implication introduction rule “bottom up:”

$$\dfrac{\begin{array}{c}\overline{\vdash (A \wedge B) \wedge C}\;x \\ \vdots \\ \vdash (A \wedge B) \wedge C \vdash B \wedge A\end{array}}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\supset I_x$$

This move adds the assumption $(A \wedge B) \wedge C$ to our local hypothesis for the duration of the proof, which we may use to finish off the proof at the top by the Ax rule. We are still obligated to fill in the missing gap between Ax and $\supset I$, but our job is now a bit easier, since we have gotten rid of the $\supset$ connective from the consequence in the goal. Next, we can try to simplify the goal again by applying the conjunction introduction rule to get rid of the $\wedge$ in the goal:

$$\dfrac{\dfrac{\begin{array}{c}\overline{\vdash (A \wedge B) \wedge C}\;x \\ \vdots \\ \vdash B\end{array} \qquad \begin{array}{c}\overline{\vdash (A \wedge B) \wedge C}\;x \\ \vdots \\ \vdash A\end{array}}{\vdash B \wedge A}\wedge I}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\supset I_x$$

We now have two sub-proofs to complete: a deduction concluding B and a deduction concluding A from our local hypothesis $(A \wedge B) \wedge C$. At this point, the consequences of our goals are as simple as they can be—they no longer contain any connectives for us to work with. Therefore, we instead switch to work “top down” from our assumptions. We are allowed to assume $(A \wedge B) \wedge C$, so let’s eliminate the unnecessary proposition C using a conjunction elimination rule in both sub-proofs:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\overline{\vdash (A \wedge B) \wedge C}\;x}{\vdash A \wedge B}\wedge E_1 \\ \vdots \\ \vdash B\end{array} \qquad \begin{array}{c}\dfrac{\overline{\vdash (A \wedge B) \wedge C}\;x}{\vdash A \wedge B}\wedge E_1 \\ \vdots \\ \vdash A\end{array}}{\vdash B \wedge A}\wedge I}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\supset I_x$$

We can now finish off the entire proof by using conjunction elimination “top down” in both sub-proofs, closing the gap between assumptions and conclusions as shown in Figure 2.2. Since there are no unjustified branches at the top of the tree (every leaf is an axiom provided by the $\supset I$ introduction rule) and there are no longer any gaps in the proof, we have completed the deduction of our goal. *End example* 2.1.

Remark 2.1. The bookkeeping that keeps track of the scope of local axioms introduced by the $\supset I$, $\vee E$, and $\exists E$ rules is important for ruling out bogus proofs that appear to be closed but manage to deduce something like $\vdash \bot$ that should be impossible. For example, we could build a closed proof of $\vdash \bot$ by using the $\supset I$ rule incorrectly as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \top}\,\top I}{\vdash \bot \supset \top}\,{\supset} I_x \qquad \dfrac{}{\vdash \bot}\,x}{\vdash (\bot \supset \top) \wedge \bot}\,\wedge I}{\vdash \bot}\,\wedge E_2$$

Notice how the local axiom x that is introduced by the $\supset I_x$ rule in the left sub-proof has been improperly "leaked" into the right sub-proof. This leak goes against the constraints of the $\supset I_x$ rule and so the above proof tree is not well-formed. Likewise, we can build another proof of $\vdash \bot$ by incorrectly applying the $\vee E$ rule as follows:

$$\dfrac{\dfrac{\dfrac{}{\vdash \top}\,\top I}{\vdash \top \vee \bot}\,\vee I_1 \qquad \dfrac{}{\vdash \bot}\,y \qquad \dfrac{}{\vdash \bot}\,y}{\vdash \bot}\,\vee E_{x,y}$$

Again, the above proof is not well-formed because the constraints of the $\vee E_{x,y}$ rule are not met: the local axiom y has been used in the middle premise but its scope is limited to only the right premise. The use of identifiers for local axiom bookkeeping is more explicit than many other presentations of natural deduction systems, but every system of natural deduction must enforce equivalent restrictions on these kinds of rules with local axioms. *End remark* 2.1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash (A \wedge B) \wedge C}\,x}{\vdash A \wedge B}\,\wedge E_1}{\vdash B}\,\wedge E_2 \qquad \dfrac{\dfrac{\dfrac{}{\vdash (A \wedge B) \wedge C}\,x}{\vdash A \wedge B}\,\wedge E_1}{\vdash A}\,\wedge E_1}{\vdash B \wedge A}\,\wedge I}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\,{\supset} I_x$$

FIGURE 2.2. NJ (natural deduction) proof of $\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)$.

Remark 2.2. The side conditions on free proposition variables in the $\forall I$ and $\exists E$ rules are perhaps the most complex ones to understand, but are nonetheless crucial for the overall logic to make sense. For example, it makes intuitive sense that if A is true for all choices of X, then there is some choice of X such that A is true. Stated formally, this intuition can be encoded into the proposition $(\forall X.A) \supset (\exists X.A)$, which can be proved in NJ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \forall X.X}\,y}{\vdash Y}\,\forall E}{\vdash \exists X.X}\,\exists I}{\vdash (\forall X.X) \supset (\exists X.X)}\,{\supset}I_y$$

The converse implication $(\exists X.A) \supset (\forall X.A)$—that if A is true for some X then it must be true for all X—does not intuitively make sense, and indeed is not provable in NJ. However, we can prove such a statement if we are sloppy with the side conditions in $\forall I$ and $\exists E$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \exists X.X}\,y \quad \dfrac{}{\vdash X}\,z}{\vdash X}\,\exists E_{X,z}}{\vdash \forall X.X}\,\forall I_X}{\vdash (\exists X.X) \supset (\forall X.X)}\,{\supset}I_y$$

This proof is *not* well-formed because the conclusion of the $\exists E_{X,z}$ rule is $\vdash X$, which contains a free occurence of X (as just plainly itself). It is fortunate that the restrictions on free proposition variables prevent a proof of $\vdash (\exists X.X) \supset (\forall X.X)$ in NJ since that leads to clearly wrong conclusions like $\vdash \bot$, similar to Remark 2.1, as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \exists X.X}\,y \quad \dfrac{}{\vdash X}\,z}{\vdash X}\,\exists E_{X,z}}{\vdash \forall X.X}\,\forall I_X}{\vdash (\exists X.X) \supset (\forall X.X)}\,{\supset}I_y \quad \dfrac{\dfrac{}{\vdash \top}\,\top I}{\vdash \exists X.X}\,\exists I}{\vdash \forall X.X}\,{\supset}E}{\vdash \bot}\,\forall E$$

End remark 2.2.

Logical harmony

Now that we know about some connectives and their rules of inference in our system of natural deduction, we would like to have some assurance that what we have defined is sensible in some way. To this end, we can insist on *logical harmony*, an idea that has roots in arguments by Dummett (1991), to justify that the inference rules are meaningful. Just like Goldilocks, we want rules that are neither too strong (leading to an inconsistent logic) nor too weak (leading to gaps in our knowledge), but are instead just right. Logical harmony for a particular connective can be broken down into two properties of that connective's inference rules: *local soundness* and *local completeness* (Pfenning & Davies, 2001).

For a single logical connective, we need to check that its inference rules are not too strong, meaning that they are *locally sound*, so that the results of the elimination rules are always justified. In other words, we cannot get out more than what we put in. Local soundness is expressed in terms of proof manipulations: a (potentially open) proof in which an introduction is immediately followed by an elimination can be simplified to a more direct proof. On the one hand, in the case of conjunction, if we follow $\wedge I$ with $\wedge E_1$, then we can perform the following reduction on the proof tree:

$$\dfrac{\dfrac{\overset{\vdots\,\mathcal{D}_1}{\vdash A} \qquad \overset{\vdots\,\mathcal{D}_2}{\vdash B}}{\vdash A \wedge B}\,\wedge I}{\vdash A}\,\wedge E_1 \qquad \succ \qquad \overset{\vdots\,\mathcal{D}_1}{\vdash A}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ stand for proofs that deduce $\vdash A$ and $\vdash B$, respectively. If we had forgotten to include the first premise $\vdash A$ in the $\wedge I$ rule, then this soundness reduction would have no proof to justify its conclusion. On the other hand, if we follow $\wedge I$ with $\wedge E_2$, then we have a similar reduction:

$$\dfrac{\dfrac{\overset{\vdots\,\mathcal{D}_1}{\vdash A} \qquad \overset{\vdots\,\mathcal{D}_2}{\vdash B}}{\vdash A \wedge B}\,\wedge I}{\vdash B}\,\wedge E_2 \qquad \succ \qquad \overset{\vdots\,\mathcal{D}_2}{\vdash B}$$

Additionally, we should ensure that the rules are not too weak, so that all the information that goes into a proof can still be accessed somehow. In this respect, we say that the inference rules for a logical connective are *locally complete* if they are strong enough to break an arbitrary (potentially open) proof ending with that connective into pieces and then put them back together again. For conjunction, this is expressed by the following proof transformation:

$$\overset{\vdots\,\mathcal{D}}{\vdash A \wedge B} \qquad \prec \qquad \dfrac{\dfrac{\overset{\vdots\,\mathcal{D}}{\vdash A \wedge B}}{\vdash A}\,\wedge E_1 \qquad \dfrac{\overset{\vdots\,\mathcal{D}}{\vdash A \wedge B}}{\vdash B}\,\wedge E_2}{\vdash A \wedge B}\,\wedge I$$

If we had forgotten the elimination rule $\wedge E_2$, then local completeness would fail because we would not have enough information to satisfy the premise of the $\wedge I$ introduction rule. As a result, the rules will still be sound but we would be unable to prove a basic tautology like $A \wedge B \supset B \wedge A$, which should hold by our intuitive interpretation of $A \wedge B$.

We also have local soundness and completeness for the inference rules of logical implication, although they require a few properties about the system as a whole. For local soundness, we can reduce $\supset I$ immediately followed by $\supset E$ as follows:

$$\dfrac{\dfrac{\begin{array}{c}\overline{\vdash A}^{\;x}\\ \vdots\;\mathcal{D}\\ \vdash B\end{array}}{\vdash A \supset B}\supset I_x \qquad \begin{array}{c}\vdots\;\mathcal{E}\\ \vdash A\end{array}}{\vdash B}\supset E \qquad \succ \qquad \begin{array}{c}\vdots\;\mathcal{E}\\ \vdash A\\ \vdots\;\mathcal{D}\{\mathcal{E}/x\}\\ \vdash B\end{array}$$

where $\mathcal{D}\{\mathcal{E}/A\}$ is the *substitution* of the proof $\mathcal{E}$ for any uses of the local axiom x in $\mathcal{D}$. The substitution gives us a modified proof that no longer needs that particular local axiom x of $\vdash A$, since any time the x axiom was used we instead place a full copy of the $\mathcal{E}$ proof of $\vdash A$. For local completeness, we can expand an arbitrary proof $\mathcal{D}$ of $\vdash A \supset B$ as follows:

$$\begin{array}{c}\vdots\;\mathcal{D}\\ \vdash A \supset B\end{array} \qquad \prec \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\;\mathcal{D}\\ \vdash A \supset B\end{array} \quad \overline{\vdash A}^{\;x}}{\vdash B}\supset E}{\vdash A \supset B}\supset I_x$$

Notice that on the right hand side the additional axiom x introduced by the use of the $\supset I_x$ introduction rule is implicitly unused in the proof $\mathcal{D}$.

The local soundness for the inference rules of logical disjunction follow from the techniques used to show soundness of both conjunction and implication: disjunction both uses a choice of two alternatives as well as a substitution for local axioms. By letting i stand for either 1 or 2, we have the following reduction for either case when $\vee I_1$ is followed by $\vee E$ or $\vee I_2$ is followed by $\vee E$:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\;\mathcal{D}\\ A_i\end{array}}{\vdash A_1 \vee A_2}\vee I_i \qquad \begin{array}{c}\overline{A_1}^{\;x_1}\\ \vdots\;\mathcal{E}_1\\ C\end{array} \qquad \begin{array}{c}\overline{A_2}^{\;x_1}\\ \vdots\;\mathcal{E}_2\\ C\end{array}}{\vdash C}\vee E_{x_1,x_2} \qquad \succ \qquad \begin{array}{c}\vdots\;\mathcal{D}\\ \vdash A_i\\ \vdots\;\mathcal{E}_i\{\mathcal{D}/x_i\}\\ \vdash E\end{array}$$

This reduction uses the same substitution operation as for the local soundness of implication, where the correct premise $\mathcal{E}_i$ is selected to match the possible choice of introduction rules. The local completeness, we can expand an arbitrary proof $\mathcal{D}$ of $\vdash A \vee B$ as follows:

$$\begin{array}{c}\vdots\,\mathcal{D}\\ \vdash A \vee B\end{array} \quad \prec \quad \dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ \vdash A \vee B\end{array} \quad \dfrac{\overline{\vdash A}^{\,x}}{\vdash A \vee B}\vee I_1 \quad \dfrac{\overline{\vdash B}^{\,y}}{\vdash A \vee B}\vee I_2}{\vdash A \vee B}\vee E_{x,y}$$

Note that this expansion may appear different from the ones that came before because the introduction rules $\vee I_1$ and $\vee I_2$ appear above the elimination rule $\vee E$ instead of below by the typographic structure of the proof tree, but still the introductions logically occur *after* the elimination by the meaning of the proof tree.

Demonstrating local soundness and completeness for the inference rules of the nullary connectives for truth and falsehood may be deceptively basic. Since there is no $\top E$ rule, local soundness of the $\top$ inference rules is trivially true: there is no possible way to have a proof where $\top I$ is followed by $\top E$ because there is no $\top E$ rule, and so local soundness is vacuous. Likewise, the local soundness of the $\bot$ inference rules is trivially true because there is no $\bot I$ rule, so soundness is again vacuous. However, we still have to demonstrate local completeness by transforming arbitrary proofs of $\vdash \top$ and $\vdash \bot$ into ones that apply all possible introduction and elimination rules for the connectives. In the case of $\top$, because the $\top I$ rule is always available, this transformation just throws away the original, unnecessary proof and replaces it with just $\top I$:

$$\begin{array}{c}\vdots\,\mathcal{D}\\ \vdash \top\end{array} \quad \prec \quad \dfrac{}{\top}\top I$$

In the case of $\bot$, because $\bot E$ only requires a proof of $\vdash \bot$ as its premise, this transformation just adds on a final $\bot E$ inference:

$$\begin{array}{c}\vdots\,\mathcal{D}\\ \vdash \bot\end{array} \quad \prec \quad \dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ \vdash \bot\end{array}}{\vdash \bot}\bot E$$

Note that both of these transformations are nullary versions of local completeness for logical conjunction and disjunction illustrated above. Therefore, we can be sure that the inference rules for $\top$ and $\bot$ are sensible.

Finally, the soundness and completeness of the quantifiers relies on the additional side conditions on their inference rules restricting the allowable free proposition variables. For the local soundness of the inference rules for universal quantification, we can reduce $\forall I$ immediately followed by $\forall E$ as follows:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\mathcal{D}\ (X \notin FV(*))\\ \vdash A\end{array}}{\vdash \forall X.A}\,\forall I_X}{A\{B/X\}}\,\forall E \qquad\succ\qquad \begin{array}{c}\vdots\,\mathcal{D}\{B/X\}\\ \vdash A\{B/X\}\end{array}$$

Note that in order to perform the reduction and get the same conclusion, we must substitute B for X in the entire proof $\mathcal{D}$. The fact that X is not free in any of the open leaves in the proof $\mathcal{D}$ (which is a required condition of the premise of $\forall I_X$) means that those leaves are left unchanged by the substitution, so that the overall fringe of the proof tree follows the same pattern. For the local completeness of the inference rules for universal quantification, we can expand an arbitrary proof $\mathcal{D}$ of $\vdash \forall X.A$ as follows:

$$\begin{array}{c}\vdots\,\mathcal{D}\ (X \notin FV(*))\\ \vdash \forall X.A\end{array} \qquad\prec\qquad \dfrac{\dfrac{\begin{array}{c}\vdots\,\mathcal{D}\ (X \notin FV(*))\\ \vdash \forall X.A\end{array}}{\vdash A}\,\forall E}{\vdash \forall X.A}\,\forall I_X$$

Note that since there are countably many proposition variables, we can pick some X which does not appear in the leaves of $\mathcal{D}$ without loss of generality since the choice doesn't matter (because we can always rename the bound X in $\forall X.A$ by α equivalence as necessary), which lets us satisfy the side condition imposed by the $\forall I_X$ rule.

The local soundness and completeness of the inference rules for existential quantification combines ideas previously seen in disjunction and universal quantification. We can reduce $\exists I$ immediately followed by $\exists E$ as follows:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ \vdash A\{B/X\}\end{array}}{\vdash \exists X.A}\,\exists I \qquad \begin{array}{c}\overline{\vdash A}^{\;x}\\ \vdots\,\mathcal{E}\ (X \notin FV(*))\\ \vdash C\end{array} \qquad (X \notin FV(C))}{\vdash C}\,\exists E_{X,x} \qquad\succ\qquad \begin{array}{c}\vdots\,\mathcal{D}\\ \vdash A\{B/X\}\\ \vdots\,\mathcal{E}\{B/X, \mathcal{D}/x\}\\ \vdash C\end{array}$$

Note how this reduction involves two different kinds of substitution: substituting the proposition B for the proposition variable X and substituting the proof $\mathcal{D}$ for the local axiom x. The side condition that X is not free in C, nor in the leaves of $\mathcal{E}$, is important to make sure that the conclusion and leaves remain the same after the substitutions. We can expand an arbitrary proof $\mathcal{D}$ of $\vdash \exists X.A$ as follows:

$$\begin{array}{c}\vdots\ \mathcal{D}\\ \vdash \exists X.A\end{array} \quad\prec\quad \dfrac{\begin{array}{c}\vdots\ \mathcal{D}\\ \vdash \exists X.A\end{array} \quad \dfrac{\dfrac{}{\vdash A}\,x}{\vdash \exists X.A}\,\exists I}{\vdash \exists X.A}\,\exists E_{X,x}$$

which follows the general pattern of the disjunction completeness expansion, but notice that the conclusion $\vdash \exists X.A$ and right sub-proof follows the extra side conditions about the free proposition variable X.

The λ-Calculus

The λ-calculus, first defined by Church in the 1930s, is a remarkably simple yet powerful model of computation. The original language of *terms* (denoted by M, N) is defined by only three parts: abstracting a program with respect to a parameter (i.e. a function term: $\lambda x.M$), reference to a parameter (i.e. a variable term: x), and applying a program to an argument (i.e. a function application term: $M\ N$). Despite this simple list of features, the untyped λ-calculus is a complete model of computation equivalent to Turing machines. It is often used as a foundation for understanding the static and dynamic semantics of programming languages as well as a platform to experiment with new language features. In particular, functional programming languages are sometimes thought of as notational convenience that desugars to an underlying core language based on the λ-calculus.

Dynamic semantics

The dynamic behavior of the λ-calculus is defined by three principles. The most basic principle is called the α *law* or α *equivalence*, and it asserts that the particular choice of names for bound variables does not matter; the defining characteristic for a variable is where it was introduced, enforcing a notion of static scope. We already saw the principle of α equivalence arise for logical quantifiers in Section 2.1, and the same idea helps understand the meaning of functions as λ-abstractions $\lambda x.M$ which bind the variable x in M. For instance, the identity function that immediately returns its argument unchanged may be written as either $\lambda x.x$ or $\lambda y.y$, both of which are considered α equivalent which is written $\lambda x.x =_\alpha \lambda y.y$. As with the logical quantifiers, we will never be more discerning of λ-calculus terms than α equivalence: if $M =_\alpha N$ then we will always treat M and N as the "same" term.

The other dynamic principles of the λ-calculus deserve a more explicit treatment because of how drastically they can alter terms. For this purpose, we will employ rules that explain how to rewrite one λ-calculus term into another. More specifically, a *rewriting rule* R, written

$$M \succ_R N$$

and pronounced "M rewrites (by R) to N," is a binary relation between terms. Rewriting rules can be combined by offering a choice between them, so that $M \succ_{RS} N$, pronounced "M rewrites (by R or S) to N," whenever $M \succ_R N$ or $M \succ_S N$. We also denote the inverse rewriting rule by flipping the direction of the $\succ$ relation symbol, so that $N \prec_R M$ exactly when $M \succ_R N$.

The second principle is called the β *law* orβ *reduction*, and it provides the primary computational force of the λ-calculus. Given a λ-abstraction (i.e. a term of the form $\lambda x.M$) that is applied to an argument, we may calculate the result by substituting the argument for every reference to the λ-abstraction's parameter:

$$(\lambda x.M)\ N \succ_\beta M\{N/x\}$$

The term $M\{N/x\}$ is notation for performing capture-avoiding substitution of the term N for the free occurrences of variable x in M, such that the static bindings of variables are preserved.[3] The third principle is called the η *law* orη *expansion*, and it imbues functions with a form of extensionality. In essence, a λ-abstraction that does nothing but forward its parameter to another function the same as that original function:

$$M \prec_\eta (\lambda x.M\ x) \qquad\qquad (x \notin FV(M))$$

Note that this rule is restricted so that M may not refer to the variable x introduced by the abstraction, denoted by the function $FV(M)$ that computes the set of free variables in v, again to preserve static binding.

[3] As before, more details about α equivalence and capture-avoiding substitution in the λ-calculus are given by Barendregt (1985) and Pierce (2002).

Even though the λ-calculus with just functions alone is sufficient for modeling all computable functions, it is often useful to enrich the language with other constructs. For instance, we may add pairs to the λ-calculus by giving a way to build a pair out of two other terms, (M, N), as well as projecting out the first and second components from a pair, $\pi_1(M)$ and $\pi_2(M)$. We may define the dynamic behavior of pairs in the λ-calculus similarly to the way we did for functions. Since pairs do not introduce any parameters, they are a bit simpler than functions. The main computational principle, by analogy called β reduction for pairs, extracts a component out of a pair when it is demanded:

$$\pi_1\,(M, N) \succ_\beta M \qquad\qquad \pi_2\,(M, N) \succ_\beta N$$

The extensionality principle, here called η expansion for pairs, expands a term M with the pair formed out of the first and second components of M:

$$M \prec_\eta (\pi_1(M), \pi_2(M))$$

Along with pairs, we can add a unit value to the λ-calculus, which is a nullary form of pair containing no elements, written $()$, that expresses a lack of any interesting information. On the one hand, since the unit value contains no elements, there are no projections out of it, and therefore it has no meaningful β reduction. On the other hand, the extensionality principle is quite strong, and the η expansion for the unit replaces any term M with the canonical unit value:

$$M \prec_\eta ()$$

This rule can be read as the nullary version of the η rule for pairs, where M did not contain any interesting information, and so it is irrelevant.

We can also add explicit choice to the λ-calculus by extending the language with (tagged, disjoint) unions, which are like boolean values that carry some extra information. First, we add the two ways to build a value of the union by tagging a term with our choice, either $\iota_1\,(M)$ or $\iota_2\,(M)$. Second, we add the method of using a tagged union by performing case analysis, $\mathbf{case}\,M\,\mathbf{of}\;\iota_1\,(x_1) \Rightarrow N_1 \mid \iota_1\,(x_2) \Rightarrow N_2$, that checks the discriminant M to pick which branch $\iota_1\,(x_1) \Rightarrow N_1$ or $\iota_2\,(x_2) \Rightarrow N_2$ to pursue. Since the term for case analysis introduces variables like function terms do, the dynamic behavior of tagged unions also relies on substitution. The main computational principle of β reduction for tagged unions checks which of the two tags were used to build the discriminant and then extracts the payload of the union by binding it to a variable within the term of the corresponding branch:

$$\begin{array}{ll} \textbf{case}\, \iota_1\,(M)\, \textbf{of} & \\ \quad \iota_1\,(x_1) \Rightarrow N_1 & \succ_\beta N_1\,\{M/x_1\} \\ \quad \iota_2\,(x_2) \Rightarrow N_2 & \end{array} \qquad \begin{array}{ll} \textbf{case}\, \iota_1\,(M)\, \textbf{of} & \\ \quad \iota_1\,(x_1) \Rightarrow N_1 & \succ_\beta N_2\,\{M/x_2\} \\ \quad \iota_2\,(x_2) \Rightarrow N_2 & \end{array}$$

The extensionality principle of η expansion for tagged unions says that every tagged union value must be constructed by one of the two possible tagging methods by expanding a term M with one that is computed by using case analysis on M to determine which tag was chosen and then returning the same payload and tag:

$$M \prec_\eta \begin{array}{l} \textbf{case}\, M\, \textbf{of} \\ \quad \iota_1\,(x_1) \Rightarrow \iota_1\,(x_1) \\ \quad \iota_2\,(x_2) \Rightarrow \iota_2\,(x_2) \end{array}$$

As before, we can add the nullary form of the binary tagged unions which represent an impossible void value: since tagged unions provide a choice of two ways to build results, there is no way to build a void result. To go along with impossible results, we also have an empty case analysis void terms, $\textbf{case}\, M\, \textbf{of}$, which will explicitly never produce any answer because a void term M cannot produce an answer. Like with units, there is no meaningful β reduction for void expressions because there is no void value for the empty case analysis to inspect. However, the extensionality principle is again strong, as it asserts that there is no value of the void type by explicitly discards any potential result a void term M might return through an empty case analysis:

$$M \prec_\eta \textbf{case}\, M\, \textbf{of}$$

This rule can be understood as the nullary version of the η rule for tagged unions, where there are no possible options for the program to proceed. Intuitively, there should be no way to encounter a void term during evaluation, since there are no ways to create void results, and so this η rule explicitly acknowledges that a void term M can only exist in a dead code branch and its results are therefore irrelevant.

Remark 2.3. A basic rewriting rule like $\succ_R$ does not necessarily confer any general properties about the relation, so we systematically denote the enrichment of a rewriting relation with useful closure properties by changing the shape of the relation symbol $\succ$. First off, we have *general R reduction*, denoted by $M \to_R N$ and pronounced "M R-reduces to N," which is the *compatible closure* of $\succ_R$ allowing for the $\succ_R$ rule to be applied in any context within M. Syntactically, a *context* (denoted by C) is a λ-calculus term with a single hole (denoted by $\Box$), and we can plug a term M into a context C (written as the operation $C[M]$) by replacing the $\Box$ in C with M. In terms of contexts, general R reduction is defined as the smallest relation $\to_R$ that includes $\succ_R$ and is closed under compatibility (*comp*) as follows:

$$\frac{M \succ_R N}{M \to_R N} \qquad\qquad \frac{M \to_R N}{C[M] \to_R C[N]}\; \mathit{comp}$$

Unlike the capture-avoiding substitution operation $M\{N/x\}$, plugging a term M into a context C might capture free variables of the term, so that even if x is free in M, x might not be free in $C[M]$. As a consequence, α equivalence *does not* commute with context filling in the same way that it commutes with capture-avoiding substitution. For example, we might say that $\lambda x.\Box =_\alpha \lambda y.\Box$, but $(\lambda x.\Box)[x] = \lambda x.x \neq_\alpha \lambda y.x = (\lambda y.\Box)[x]$.

Next up, we have the *R reduction theory* (or *R rewriting theory*), denoted by $M \twoheadrightarrow_R N$, which is the *reflexive-transitive closure* of $\to_R$ allowing for zero or more repetitions of $\twoheadrightarrow_R$ reductions. The R reduction theory is defined as the smallest relation $\twoheadrightarrow_R$ that includes $\to_R$ and is closed under reflexivity (*refl*) and transitivity (*trans*) as follows:

$$\frac{M \to_R N}{M \twoheadrightarrow_R N} \qquad\qquad \frac{}{M \twoheadrightarrow_R M}\; \mathit{refl} \qquad\qquad \frac{M \twoheadrightarrow_R M' \quad M' \twoheadrightarrow_R N}{M \twoheadrightarrow_R N}\; \mathit{trans}$$

Note that above definition of $\twoheadrightarrow_R$ is the same as taking the compatible-reflexive-transitive closure of $\succ_R$ directly.

For the most generality, we have the *R equational theory*, denoted by $M =_R N$ and pronounced as "M R-equals N," which is the *symmetric-transitive closure* of $\twoheadrightarrow_R$ that allows for reductions to be applied in both directions as many times as desired. The R equational theory is defined as the smallest relation $=_R$ that includes $\twoheadrightarrow_R$ and is closed under symmetry (*symm*) and transitivity (*trans*) as follows:

$$\frac{M \twoheadrightarrow_R N}{M =_R N} \qquad \frac{N =_R M}{M =_R N}\ \mathit{symm} \qquad \frac{M =_R M' \quad M' =_R N}{M =_R N}\ \mathit{trans}$$

Note that the above definition of $=_R$ is the same as taking the compatible-reflexive-symmetric-transitive closure of $\succ_R$ directly.

Finally, we have R *operational reduction*, denoted by $M \mapsto_R N$, which gives us the R *operational semantics*, denoted by $M \mapsto\!\!\!\!\twoheadrightarrow_R N$, as the reflexive-transitive closure of $\mapsto_R$. Both of these are restrictions on the above more general reduction relations: R operational reduction is a limited form of general R reduction and the R operational semantics is a limited form of the R reduction theory. The purpose of the operational semantics is to specify how programs are to be executed by specifying a clear order on when each reduction step of the program occurs; there should be enough possible reductions to reach a result, but not so many that there are gratuitously many choices for what to do at every step. This ordering for selecting the next reduction step can be achieved by restricting compatibility, which allowed reduction to occur in any context, to only allowing reduction to occur in a specially chosen subset of contexts called *evaluation contexts*, usually denoted by the variable E. Given a choice of evaluation contexts, R operational reduction and the R operational semantics are defined as the smalled relations $\mapsto_R$ and $\mapsto\!\!\!\!\twoheadrightarrow_R$ closed under the following rules:

$$\frac{M \succ_R N}{E[M] \mapsto_R E[N]}\ \mathit{eval} \quad \frac{M \mapsto_R N}{M \mapsto\!\!\!\!\twoheadrightarrow_R N} \quad \frac{}{M \mapsto\!\!\!\!\twoheadrightarrow_R M}\ \mathit{refl} \quad \frac{M \mapsto\!\!\!\!\twoheadrightarrow_R M' \quad M' \mapsto\!\!\!\!\twoheadrightarrow_R N}{M \mapsto\!\!\!\!\twoheadrightarrow_R N}\ \mathit{trans}$$

Since we have to make a choice for which contexts are evaluation contexts, there can be many possible operational semantics for a given language. As an example, we can define a *call-by-name* operational semantics $\mapsto\!\!\!\!\twoheadrightarrow_\beta$ for our λ-calculus discussed so far by choosing the following evaluation contexts:

$$\begin{aligned} E \in \mathit{EvalCxt} ::= {} & \square \mid \pi_1(E) \mid \pi_2(E) \mid E\ N \\ & \mid (\mathbf{case}\, E\, \mathbf{of}\ ) \mid (\mathbf{case}\, E\, \mathbf{of}\, \iota_1\,(x) \Rightarrow N_1 \mid \iota_2\,(y) \Rightarrow N_2) \end{aligned}$$

by using the family of operational β laws.

As with a basic rewriting rule $\succ_R$, we denote the inverse of the directed reduction relations $\to_R$, $\twoheadrightarrow_R$, $\mapsto_R$, and $\mapsto\!\!\!\!\twoheadrightarrow_R$ by flipping the direction of the arrow, so that $N \leftarrow_R$

M if and only if $M \to_R N$ and so on. Since the equational theory $=_R$ is symmetric, it is undirected, so it is its own inverse. *End remark* 2.3.

Static semantics

So far, we have only considered the dynamic meaning of the λ-calculus without any mention of its static properties. In particular, now that we have both functions and pairs, we may want to statically check and rule out programs that might "go wrong" during calculation. For instance, if we apply a pair to an argument, $(x, y)\ z$, then there is nothing we can do to reduce this program any further. Likewise, it is nonsensical to ask for the second component of a function, $\pi_2(\lambda x.x)$. We may rule out such ill-behaved programs by using a *type system* which guarantees that such situations never occur by assigning a type to every term and ensuring that programs are used in accordance to their types. For instance, we may give a function type, $A \to B$, to λ-abstractions as follows:

$$\dfrac{\begin{array}{c}\overline{x : A}^{\;x}\\ \vdots \\ M : B\end{array}}{\lambda x.M : A \to B}\,{\to}I_x$$

where $\lambda x.M : A \to B$ means that the function $\lambda x.M$ has type $A \to B$. The premise to this rule requires that M has type B assuming that all free occurrences of x in M have type A. Since the variable x bound in the conclusion, it is closed off by the premise of the rule because the type values that x can stand for in M has nothing to do with any other x that might occur elsewhere in a larger term. Having given a rule for introducing a term of function type, we can now restrict application to only occur for terms of the correct type:

$$\dfrac{M : A \to B \quad N : A}{M\ N : B}\,{\to}E$$

This rule ensures that if we apply a term M to an argument, then M must have a function type.

Likewise, we may give a product type, $A \times B$, to the creation of a pairs

$$\dfrac{M : A \quad N : B}{(M, N) : A \times B}\,{\times}I$$

as well as limiting first and second projection to terms of a product type:

$$\frac{M : A \times B}{\pi_1(M) : A}\ {\times}E_1 \qquad\qquad \frac{M : A \times B}{\pi_2(M) : B}\ {\times}E_2$$

The unit type, 1, is a degenerate form of product types with a single canonical value

$$\frac{}{() : 1}\ 1I$$

and no other typing rules.

Tagged unions belong to sum types, $A + B$, which has two different rules for the creation of the two distinctly tagged values:

$$\frac{M : A}{\iota_1\,(M) : A + B}\ {+}I_1 \qquad\qquad \frac{M : B}{\iota_2\,(M) : A + B}\ {+}I_2$$

The case analysis term for sum types has the most complex rule, requiring three premises (one for the discriminant and two for the branches), two of which bind variables which appear free in their respective sub-terms just like in the rule for λ-abstractions:

$$\frac{M : A + B \quad \begin{matrix}\overline{x_1 : A}^{\;x_1} \\ \vdots \\ N_1 : C\end{matrix} \quad \begin{matrix}\overline{x_2 : B}^{\;x_2} \\ \vdots \\ N_2 : C\end{matrix}}{(\mathbf{case}\, M\, \mathbf{of}\ \iota_1\,(x_1) \Rightarrow N_1 \mid \iota_1\,(x_2) \Rightarrow N_2) : C}\ {+}E_{x_1,x_2}$$

This rule says that a case analysis expression on a term M with the sum type $A + B$ has a result of type C if the terms N_1 and N_2 in both branches have the type C, under the assumption that all free occurrences of x_1 in N_1 has type A and all free occurrences of x_2 in N_2 has type B. The void type, 0, is a degenerate form of sum types with no possible values and one case analysis term following the typing rule

$$\frac{M : 0}{\mathbf{case}\, M\, \mathbf{of}\ \ : C}\ 0E$$

which says that the result of an empty case analysis on a term M of type 0 can be said to have any type C because there will never be any result.

With all these rules in place, nonsensical programs like $\pi_1(\lambda x.x)$ are now ruled out, since they cannot be given a type. The static semantics (i.e. the typing rules) and the dynamic semantics (i.e. the reduction and expansion relationships) of this simply typed λ-calculus are summarized in Figure 2.3. Note that the η laws, if left unchecked, have the potential to cause unwanted relationships between terms. The different ways that η has the potential to cause problems can be very subtle (Klop & de Vrijer, 1989), but the issue is most clearly seen for units. In particular, η^1 expansion for units says that *any* term can be replaced with unit value $()$. But this apparently far-reaching law is clearly nonsensical for representing programs: if every possible program is just $()$ then there's no point in evaluating anything because there is never an interesting answer! The other direction is not much better; η^1 reduction says that the unit value $()$ could just as well become anything else, leading to many different conflicting answers whenever we encounter a unit value.

This conundrum is somewhat self-imposed, however: clearly the η^1 law shouldn't apply to *every* term, but only to terms we expect will result in a unit value anyway. Therefore, the η laws are all restricted to apply only to terms of an appropriate type, so for example the η^1 law only expands terms of type 1 with $()$. This creates an interesting split in the relationships between terms, where we have the β laws that do not depend on types, so that they still make sense for reasoning about untyped terms, in contrast with the η laws that do depend on types to make sense, so that they require typing information to ensure that they are correctly applied.

Remark 2.4. We should note that some care needs to be taken during a type derivation to make sure that the distinction between variables in different scopes is clear. For example, consider the following typing derivation of the function $\lambda x.\lambda x.x$:

$$\dfrac{\dfrac{\dfrac{}{x : A}\, x}{\lambda x.x : B \to A}\,{\to}I_x}{\lambda x.\lambda x.x : A \to B \to A}\,{\to}I_x$$

This typing derivation is not valid! In particular, note that the function $\lambda x.\lambda x.x$ is α equivalent to $\lambda x.\lambda y.y$ by renaming the second bound variable, which represents a binary function that returns its second argument. The problem is that by rebinding the same variable x within the same scope, it is easy to have confusion about which of the two arguments is meant when referring to x. This is why typing rules like $\to I$ for terms which bind variables introduce a new scope in their premise to prevent this

$$\begin{aligned}
X,Y,Z \in \mathit{TypeVariable} &::= \dots\\
A,B,C \in \mathit{Type} &::= X \mid 1 \mid 0 \mid A\times B \mid A+B \mid A\to B\\
x,y,z \in \mathit{Variable} &::= \dots\\
M,N \in \mathit{Term} &::= x \mid () \mid \mathbf{case}\,M\,\mathbf{of}\\
&\qquad \mid (M,N) \mid \pi_1(M) \mid \pi_2(M)\\
&\qquad \mid \iota_1\,(M) \mid \iota_2\,(M) \mid (\mathbf{case}\,M\,\mathbf{of}\,\iota_1\,(x)\Rightarrow N_1 \mid \iota_2\,(y)\Rightarrow N_2)\\
&\qquad \mid \lambda x.M \mid M\ N\\
H,J \in \mathit{Judgement} &::= M : A
\end{aligned}$$

$$\frac{}{():1}\,1I \qquad \text{no } 1E \text{ rule} \qquad \text{no } 0I \text{ rule} \qquad \frac{M:0}{\mathbf{case}\,M\,\mathbf{of}\ :C}\,0E$$

$$\frac{M:A \quad N:B}{(M,N):A\times B}\,\times I \qquad \frac{M:A\times B}{\pi_1(M):A}\,\times E_1 \qquad \frac{M:A\times B}{\pi_2(M):B}\,\times E_2$$

$$\frac{M:A}{\iota_1\,(M):A+B}\,+I_1 \qquad \frac{M:B}{\iota_2\,(M):A+B}\,+I_2 \qquad \frac{M:A+B \quad \begin{array}{c}\overline{x:A}^{\;x}\\ \vdots\\ N_1:C\end{array} \quad \begin{array}{c}\overline{y:B}^{\;y}\\ \vdots\\ N_2:C\end{array}}{\mathbf{case}\,M\,\mathbf{of}\,\iota_1\,(x)\Rightarrow N_1|\iota_2\,(y)\Rightarrow N_2 : C}\,+E_{x,y}$$

$$\frac{\begin{array}{c}\overline{x:A}^{\;x}\\ \vdots\\ M:B\end{array}}{\lambda x.M:A\to B}\,\to I_x \qquad \frac{M:A\to B \quad N:A}{M\ N:B}\,\to E$$

$$\begin{array}{llll}
(\beta^1) & \text{no rule} & (\eta^1) & M:1 \prec_\eta ()\\
(\beta^0) & \text{no rule} & (\eta^0) & M:0 \prec_\eta \mathbf{case}\,M\,\mathbf{of}\\
(\beta^\times) & \pi_i(M_1,M_2) \succ_\beta M_i & (\eta^\times) & M:A{\times}B \prec_\eta (\pi_1(M),\pi_2(M))\\
(\beta^+) & \begin{array}{l}\mathbf{case}\,\iota_i\,(M)\,\mathbf{of}\\ \quad \iota_1\,(x_1)\Rightarrow N_1\\ \quad \iota_2\,(x_2)\Rightarrow N_2\end{array} \succ_\beta N_i\,\{M/x_i\} & (\eta^+) & M:A{+}B \prec_\eta \begin{array}{l}\mathbf{case}\,M\,\mathbf{of}\\ \quad \iota_1\,(x_1)\Rightarrow \iota_1\,(x_1)\\ \quad \iota_2\,(x_2)\Rightarrow \iota_2\,(x_2)\end{array}\\
(\beta^\to) & (\lambda x.M)\ N \succ_\beta M\,\{N/x\} & (\eta^\to) & M:A{\to}B \prec_\eta \lambda x.M\ x \qquad (x\notin FV(M))
\end{array}$$

FIGURE 2.3. The simply typed λ-calculus: with unit (1), void (0), product (×), sum (+), and function (→) types.

confusion. In particular, the typing derivation for the sub-term $\lambda x.x$ is:

$$\dfrac{\dfrac{}{x : B}\,x}{\lambda x.x : B \to B}\;{\to}I_x$$

In this derivation, the variable x is already closed off, because it is bound by the λ-abstraction in the conclusion. Therefore, when we continue the derivation to type the outer λ-abstraction, the type of the bound reference of x is already fixed, and cannot be changed as in

$$\dfrac{\dfrac{\dfrac{}{x : B}\,x}{\lambda x.x : B \to B}\;{\to}I_x}{\lambda x.\lambda x.x : A \to B \to B}\;{\to}I_x$$

which is the correct typing derivation for this term. *End remark* 2.4.

Example 2.2. For an example of how to program in the λ-calculus, consider the following function which takes a nested pair, of type $(A \times B) \times C$, and swaps the inner first and second components, while discarding the outer component:

$$\lambda x.\,(\pi_2(\pi_1(x)), \pi_1(\pi_1(x)))$$

We can check that this function is indeed well-typed, using the typing rules given in Figure 2.3, by the constructing the typing derivation in Figure 2.4. Notice how the derivation bears a close structural resemblance to the proof of $\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)$ given in Figure 2.2 of Example 2.1. In addition, we can check that this function behaves as intended by applying it to a nested pair,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : (A \times B) \times C}\,x}{\pi_1(x) : A \times B}\,{\times}E_1}{\pi_2(\pi_1(x)) : B}\,{\times}E_2 \qquad \dfrac{\dfrac{\dfrac{}{x : (A \times B) \times C}\,x}{\pi_1(x) : A \times B}\,{\times}E_1}{\pi_1(\pi_1(x)) : A}\,{\times}E_1}{(\pi_2(\pi_1(x)), \pi_1(\pi_1(x))) : B \times A}\,{\times}I}{\lambda x.\,(\pi_2(\pi_1(x)), \pi_1(\pi_1(x))) : ((A \times B) \times C) \to (B \times A)}\;{\to}I_x$$

FIGURE 2.4. Typing derivation of the λ-calculus term $\lambda x.\,(\pi_2(\pi_1(x)), \pi_1(\pi_1(x)))$.

$((M_1, M_2), M_3)$, and evaluating it with the reductions given in Figure 2.3:

$$\begin{aligned}
&(\lambda x.\, (\pi_2(\pi_1(x)), \pi_1(\pi_1(x))))\ \ ((M_1, M_2), M_3) \\
&\to_{\beta\to}\ (\pi_2(\pi_1((M_1, M_2), M_3)), \pi_1(\pi_1((M_1, M_2), M_3))) \\
&\twoheadrightarrow_{\beta\times}\ (\pi_2(M_1, M_2), \pi_1\,(M_1, M_2)) \\
&\twoheadrightarrow_{\beta\times}\ (M_2, M_1)
\end{aligned}$$

which confirms that this is the function we wanted. *End example* 2.2.

Type abstraction

If we only stick to typed terms, then the language we have described so far is rather rigid and painful to use because every term must have a fixed specific type even if it doesn't matter. For example, the identity function $\lambda x.x$, which just returns its given input, works uniformly for values of any type. However, it must be given a single type like $\mathsf{Int} \to \mathsf{Int}$ or $\mathsf{String} \to \mathsf{String}$, meaning that the integer and string identity functions must be defined separately even though their definition is the same. Statically typed programming languages combat this useless redundancy with features called *polymorphism* or *generics* that correspond to *universal types* in the λ-calculus, which has been co-discovered in Girard's (1971) system F and Reynolds's (1974) polymorphic λ-calculus. The main idea is to let generic terms abstract over type variables, so that we have the term $\Lambda X.M$ similar to the λ-abstractions that represent functions, and to specialize generic terms to specific types, so that we have the term $M\ A$ similar to function application. The computational β reduction for polymorphism also mimics functions by substituting the specialized type for the abstracted type variable:

$$(\Lambda X.M)\ A \succ_\beta M\,\{A/X\}$$

Likewise, the extensional η expansion for polymorphism says that a generic term that just immediately specializes another generic term M with its applied type is the same as M:

$$M \prec_\eta \Lambda X.M\ X$$

These generic terms can be given a universal type of the form $\forall X.A$. Specialization of generic terms just involves plugging in the applied type for the variable X in the result, but the typing rule for abstraction is more tricky:

$$\frac{\begin{array}{c}\vdots\ (X \notin FV(*))\\ M : A\end{array}}{\Lambda X.M : \forall X.A}\,\forall I_X \qquad\qquad \frac{M : \forall X.A}{M\ B : A\{B/X\}}\,\forall E$$

The $\forall I$ rule imposes a side condition on its premise, $X \notin FV(*)$, which says that the type variable X cannot appear in the type of any free variable of M. With universal types, we can finally give a single, polymorphic definition of the identity function once and for all, $\Lambda X.\lambda x.x : \forall X.X \to X$, which is typed as follows:

$$\dfrac{\dfrac{\dfrac{}{x : X}\,x}{\lambda x.x : X \to X}\,{\to}I_x}{\Lambda X.\lambda x.x : \forall X.X \to X}\,\forall I_X$$

There is another complementary form of type abstraction with a very different purpose in programming languages. For the sake of supporting more modular programs, many typed languages allow for modules or other basic program units to *hide* some of their representation. That way, the implementor of the module may use details of its representation, but users of the module can only see the public interface do not have access to these private details since peaking into the private details of a module's implementation would break the abstraction and prevent the user code from linking with a different implementation. For example, we might have a module for integer sets with four components in its public interface: the empty set, a function for creating the singleton set of a given integer, a union function, and a membership function that decides if an integer is in the set. Now there are many different ways that a program could represent integer sets—arrays, linked lists, hash tables, balanced trees, higher-order functions, etc.—but the code which uses integer sets should be independent of the implementors choice of representation so that it can plug in with several different implementations of the same public interface. This type of abstraction can be modeled by *existential types* that make a choice of type private to a small fragment of the overall program. For our example of integer sets, their interface is described by the type

$$\exists X.X \times (\mathsf{Int} \to X) \times (X \to X \to X) \times (\mathsf{Int} \to X \to \mathsf{Bool})$$

where the $\exists$ abstracts over a private type denoted by the variable X, and the four components of the public interface are given by the four components of the product: the empty set of type X, the singleton function of type $\mathsf{Int} \to X$, the union function of type $X \to X \to X$, and the membership function of type $\mathsf{Int} \to X \to \mathsf{Bool}$.

How do we write programs with existential types? To be explicit about when we are abstracting over a private type A used within a term M, we can package them together as $A \mathbin{@} M$ where the term is tagged with its private type. We can then use a packaged term by employing a new form of case analysis, $\textbf{case}\, M \,\textbf{of}\, X \mathbin{@} y \Rightarrow N$, which locally unpacks M and separates out its private type (bound to the type variable X) from the contents (bound to the variable y) for the purpose of evaluating the result of N. The computational β reduction for existential types unpacks a type-packaged term $A \mathbin{@} M$ that is in the eye of case analysis, substituting the concrete type A and the implementation M for the abstract type variable X and the reference x within their local scope:

$$\textbf{case}\, A \mathbin{@} M \,\textbf{of}\, X \mathbin{@} x \Rightarrow N \succ_\beta N\,\{A/X, N/x\}$$

The extensional η principle for existential types says that every value of an existential type must be a type-packaged value by expanding an existential term M into one that is computed by unpacking M to extract its private type and value, only to return a new package with the same type and value:

$$M \prec_\eta \textbf{case}\, M \,\textbf{of}\, X \mathbin{@} x \Rightarrow X \mathbin{@} x$$

This form of existential type abstraction for packages can be enforced with the following typing rules:

$$\dfrac{M : A\{B/X\}}{B \mathbin{@} M : \exists X.A}\,\exists I \qquad \dfrac{M : \exists X.A \quad \begin{array}{c}\overline{x : A}^{\,x}\\ \vdots\ (X \notin FV(*))\\ N : C\end{array} \qquad (X \notin FV(C))}{(\textbf{case}\, M \,\textbf{of}\, X \mathbin{@} x \Rightarrow N) : C}\,\exists E_{X,x}$$

To form a new package $B \mathbin{@} M : \exists X.A$, we only need to check that the underlying term M does indeed implement a program of type A with the chosen type B substituted for X. Unpacking a type abstraction is more complex, as we need to ensure that the hidden type information cannot “leak” outside its scope. Therefore, the generic

$$\begin{aligned} A, B, C \in \mathit{Type} &::= \dots \mid \forall X.A \mid \exists X.A \\ M, N \in \mathit{Term} &::= \dots \mid \Lambda X.M \mid M\ A \mid A \mathbin{@} M \mid \mathbf{case}\, M\, \mathbf{of}\, X \mathbin{@} x \Rightarrow N \end{aligned}$$

$$\dfrac{\begin{array}{c} \vdots\ X \notin FV(*) \\ M : A \end{array}}{\Lambda X.M : \forall X.A}\ \forall I_X \qquad\qquad \dfrac{M : \forall X.A}{M\ B : A\{B/X\}}\ \forall E$$

$$\dfrac{M : A\{B/X\}}{B \mathbin{@} M : \exists X.A}\ \exists I \qquad\qquad \dfrac{M : \exists X.A \quad \begin{array}{c} \overline{x : A}\ x \\ \vdots\ X \notin FV(*) \\ N : C \end{array} \quad X \notin FV(C)}{\mathbf{case}\, M\, \mathbf{of}\, X \mathbin{@} x \Rightarrow N : C}\ \exists E_{X,x}$$

$$\begin{array}{llll} (\beta^{\forall}) & (\Lambda X.M)\ A \succ_{\beta} M\,\{A/X\} & (\eta^{\forall}) & M : \forall X.A \prec_{\eta} \Lambda X.M\ X \quad (X \notin FV(M)) \\ (\beta^{\exists}) & \begin{array}{c}\mathbf{case}\, A \mathbin{@} M\, \mathbf{of} \\ X \mathbin{@} x \Rightarrow N\end{array} \succ_{\beta} N\,\{A/X, M/x\} & (\eta^{\exists}) & M : \exists X.A \prec_{\eta} \begin{array}{c}\mathbf{case}\, M\, \mathbf{of} \\ X \mathbin{@} x \Rightarrow X \mathbin{@} x\end{array} \end{array}$$

FIGURE 2.5. The polymorphic λ-calculus (i.e. system F): extending the simply typed λ-calculus with universal ($\forall$) and existential ($\exists$) type abstraction.

type variable X that is brought into scope by the case analysis cannot appear in the types of any other free variables (besides the corresponding variable x) in its scope. Additionally, the generic type X bound by the unpacking case analysis cannot appear in the return type C, which is the other source of potential leak.

The static and dynamic semantics of the universal ($\forall$) and existential ($\exists$) forms of type abstraction are summarized in Figure 2.5, which extends the simply typed λ-calculus from Figure 2.3 to be a full-fledged model of statically typed (functional) programming languages.

Proofs as Programs

Amazingly, despite their different origins and presentations, both the systems have a close, one-for-one correspondence to each other. Example 2.1 and Example 2.2 correspond to different ways of expressing the same idea. Both natural deduction and the λ-calculus end up revealing the same underlying ideas in different ways. The propositions of natural deduction are isomorphic to the types of the λ-calculus, where conjunctions are the same as pair types, disjunctions are the same as sum types, implications are the same as function types, logical truth and falsehood is the same as the unit and void types, and the two quantifiers are the same in both systems. Furthermore, the proofs of natural deduction are isomorphic to the (typed) terms of the λ-calculus. This structural similarity between the two systems gives us the slogan, "proofs as programs and propositions as types." From this point of view, natural deduction may be seen as the essence of the type system for the λ-calculus and the λ-calculus may be seen as a more concise term language for expressing proofs in natural deduction. For this reason, we may say that the λ-calculus is a natural deduction language.

The correspondence between these two systems is not just between their syntax and static structures, but also extends to the dynamic properties as well. Local soundness and completeness in natural deduction are exactly the same as the β and η laws of terms in the λ-calculus, respectively, for all the discussed types: functions, products, sums, unit, void, universal, and existential types. Therefore, it is no coincidence that the β and η rules for functions in the λ-calculus appeared as they originally did, or that conjunction and disjunction have their given introduction and elimination rules in NJ. Effectively, both the study of logic and the study of computability have lead mathematicians to (re)discover different perspectives of the same essential phenomena (Wadler, 2015).

Surprisingly, there is also a third entity in this correspondence: an algebraic structure known as *Cartesian closed categories* (Lambek & Scott, 1986). In general, a category is made up of:

- some objects A, B and C ("points"),
- some morphisms between those objects ("arrows"), a morphism f from A to B is written $f : A \to B$,
- a trivial morphism from every object to itself ("identity"), and
- the ability to chain together any two morphisms passing through the same object ("composition"). Given $f : A \to B$ and $g : B \to C$ then $g \circ f$ is a morphism from A to C ,

along with some laws about identity and composition. And Cartesian closed categories in particular are also guaranteed to have some special objects: a terminal object 1, a product object $A \times B$ for any objects A and B, and an exponential object B^A for any objects A and B. As it turns out, the terminal (1), product ($A \times B$), and exponential (B^A) objects correspond to unit (1), pair ($A \times B$), and function ($A \to B$) types in the λ-calculus and to truth ($\top$), conjunction ($A \wedge B$), and implication ($A \supset B$) in natural deduction, respectively. Cartesian closed categories may be seen as a variable-free presentation of the λ-calculus, where λ-abstractions (which bind variables) are replaced by primitive functions. Furthermore, the categorical concept of the initial object (0) and sums of objects ($A+B$) correspond with the empty (0) and sum ($A+B$) types and with logical falsehood ($\bot$) and disjunction ($A \vee B$), respectively. Since the same idea has been stumbled upon three different times from three different angles, the connection between proofs and programs cannot be a simple coincidence.

A Critical Look at the λ-Calculus

The Curry-Howard isomorphism lead to striking discoveries and developments that likely would not have arisen otherwise. The connection between logic and programming languages led to the development of mechanized proof assistants, notably the Coq system (Coquand, 1985), which are used in both the security and verification communities for validating the correctness of programs. The connection between category theory and programming languages suggested a new compilation technique for ML (Cousineau *et al.*, 1987). However, let us now look at the λ-calculus with a more critical eye. There are some defining principles and computational phenomena that are important to programming languages, but are not addressed by the λ-calculus. For example, what about:

– *Duality?* The concept of duality is important in category theory where it comes for free as a consequence of the presentation. Since the morphisms in category theory have a direction, we can just “flip all the arrows” to find its dual without any effort or creativity on our part. This action gives us a straightforward method to find the dual of any category or diagram. For example, consider the diagram that describes products in categorical terms:

$$\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[d, dashed, "!(f{,}g)"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}$$

Here, for any two objects, A and B, and morphisms, f, and g, there is the product $A \times B$ object with the projection morphisms π_1 and π_2 *out* of the product and a unique morphism *into* the product. The description of sums pops out for free by just turning that diagram around:

$$\begin{tikzcd}
A \arrow[r, "\iota_1"] \arrow[dr, "f"'] & A + B \arrow[d, dashed, "{![f,g]}"] & B \arrow[l, "\iota_2"'] \arrow[dl, "g"] \\
& C &
\end{tikzcd}$$

Now, the two projections have become two injections, ι_1 and ι_2, *into* the sum object and we have a unique morphism *out* of the sum for any f and g.

Duality also appears in logic, for example in the traditional De Morgan laws like $\neg(A \vee B) = (\neg A) \wedge (\neg B)$. Predictably, the corresponding concept of a sum object (the dual of a product) in logic is disjunction (the dual of a conjunction). If we look at the rules of NJ from Figure 2.1, the introduction rules for $A \vee B$ bear a resemblance to the elimination rules for $A \wedge B$: one is just flipped upside-down from the other. However, the elimination rule for disjunction is quite different from the introduction for conjunction. This dissimilarity comes from the asymmetry in natural deduction. We may have many premises, but only a single conclusion. It seems like a more symmetrical system of logic would be easier to methodically determine duality just like in category theory.

Likewise, this form of duality is not readily apparent in the λ-calculus. Since the λ-calculus is isomorphic to NJ, it shares the same biases and lack of symmetry. The emphasis of the language is entirely on the *production* of information: a λ-abstraction *produces* a function, a function application *produces* the result, *etc.* For this reason, the relationship between a pair, (M, N), and case analysis on tagged unions, $\textbf{case}\, M\, \textbf{of}\, \iota_1\,(x) \Rightarrow N_1 | \iota_2\,(y) \Rightarrow N_2$, is not entirely obvious. For this reason, we would like to study a language which expresses duality "for free," and which corresponds to a more symmetrical system of logic.

- *Evaluation strategy?* Reynolds (1998) observed that while functional or applicative languages may be based on the λ-calculus, the true λ-calculus implies a lazy (call-by-name) evaluation order, whereas many languages are evaluated by a strict (call-by-value) order that first reduces arguments before performing a function call.

To resolve this mismatch between the λ-calculus and strict programming languages, Plotkin (1975) defined a call-by-value variant of the λ-calculus along with a *continuation-passing style* (CPS) transformation that embeds the evaluation order into the program itself. Sabry & Felleisen (1993) give a complete set of equations for reasoning about the call-by-value λ-calculus based on Fischer's (1993) call-by-value CPS transformation, and which corresponds to Moggi's (1989) computational λ-calculus. The equations were later refined into a complete theory for call-by-value reduction by Sabry & Wadler (1997). More recently, there has been work on a theory for reasoning about call-by-need evaluation of the λ-calculus (Ariola *et al.*, 1995; Ariola & Felleisen, 1997; Maraist *et al.*, 1998), which is the strategy commonly employed by Haskell implementations, and the development of Levy's (2001) call-by-push-value framework which includes both call-by-value and call-by-name evaluation but not call-by-need.

Different evaluation strategies used by implementations of functional programming languages have been studied as different versions of the λ-calculus that embody the implementation, including various calculi for call-by-value (Plotkin, 1975; Moggi, 1989; Sabry & Felleisen, 1993; Sabry & Wadler, 1997) and call-by-need (Ariola *et al.*, 1995; Ariola & Felleisen, 1997; Maraist *et al.*, 1998) evaluation.

What we would ultimately want is not just another calculus, but instead a framework that gives a clear justification of the evaluation strategies found in programming languages, and where the relationships between strategies can be naturally expressed. Can we have a logical foundation for programming languages that is naturally strict, in the same way that the λ-calculus is naturally lazy? And which readily accounts for programs that utilize more than one evaluation strategy in the same language? Can we express the duality between evaluation strategies Filinski (1989); Curien & Herbelin (2000); Wadler (2003) generically, between arbitrarily many pairs of strategies?

– *Object-oriented programming?* The object-oriented paradigm has become a prominent part of the mainstream programming landscape. Unfortunately, what is meant by an "object" in the object-oriented sense is fuzzy, since the exact details of "what is an object" depend on choices made by the particular programming language. One concept of objects that is universal across every programming language is *dynamic dispatch* which is used to select the behavior of a method call based on the value or type of an object. Dynamic dispatch is emphasized by Kay (1993) in the form of *message passing* in the design of Smalltalk. Abadi & Cardelli (1996) give a theoretical formulation for the many features of object-oriented languages, wherein dynamic dispatch plays a central role. Can we give an account of the essence of objects, and in particular messages and dispatch, that is connected to logic and category theory in the same way as the λ-calculus? Even more, can this foundation for objects refer back to basic principles discovered independently in the field of logic?

- *Control flow?* Every programming language has some concept of control flow which can describe the order that instructions are executed, the flow of data dependencies between parts of a program, or the call-and-return protocol of functions. The λ-calculus serves as a wonderful formalization of for pure functions. However, many languages include additional computational effects, like exceptions, that let programs manipulate control flow in ways not possible with pure functions, and so they lie outside of the expressive power of the λ-calculus (Felleisen, 1991). For example, Scheme (Kelsey *et al.*, 1998) is a language based on the λ-calculus that nonetheless has operators like callcc that reifies control flow as a first-class object, which follows a traditional approach for representing control flow by adding new primitives to the λ-calculus.

  Instead, we would rather understand the flow of control in a setting where it is naturally expressed as a consequence of the language, rather than added on as an afterthought. Surprisingly, certain programmatic manipulations of control flow, like Scheme's callcc, correspond to axioms of classical logic (Griffin, 1990; Ariola & Herbelin, 2003). Since these classical reasoning principles are a well established part of logic, can we also have a corresponding language with a naturally classical representation of control as a first-class citizen?

With the aim of answering each of these questions, we will put the λ-calculus aside and we look to another logical framework instead of natural deduction. Most surprisingly, we do not have to look very far, since Gentzen (1935a) introduced the sequent calculus along side natural deduction as an alternative system of formal logic. Gentzen developed sequent calculus in order to better understand the properties of natural deduction. Therefore, to answers these questions about programming, we will look for the computational interpretation of the sequent calculus and its corresponding programming language.

CHAPTER III

Sequent Calculus

Natural deduction is not an only child; it was born with a twin sibling called the *sequent calculus*. One of Gentzen's (1935a) ground-breaking insights with the sequent calculus is the use of its namesake sequents to organize the information we have about the various propositions in question. In its most general form, a *sequent* is a conditional conglomeration of propositions:

$$A_1, A_2, \dots, A_n \vdash B_1, B_2, \dots, B_m$$

pronounced "A_1, A_2, ..., and A_n entail B_1, B_2, ..., or B_m," which states that assuming *each* of $A_1, A_2, \dots, A_n$ are true then at least *one* of $B_1, B_2, \dots, B_m$ must be true. The turnstyle ($\vdash$) in the middle of the sequent separates the *hypotheses* on the left, which we collectively write as Γ, from the *consequences* on the right, which we collectively write as Δ.

This separation between the left and right sides of the sequent gives the essential skeletal structure of the sequent calculus as a logic. As special cases, we can form several basic judgements about logical propositions using our above interpretation of the meaning of sequents by observing that an empty collection of hypotheses denotes "true" and an empty collection of consequences denotes "false." A single consequence without hypotheses $\vdash A$ means "A is true,"[1] a single hypothesis without consequences $A \vdash$ means "A is false," and the empty sequent $\vdash$ is a primitive contradiction "true entails false." So already, the basic structure of the sequent gives us a language for speaking about truth, falsehood, and contradiction without knowing anything else about the logic at hand.

[1] Note how sequents gracefully extend the single judgement $\vdash A$ of the NJ system of natural deduction, which only directly asserts the truth of propositions, so that statements of falsehood or contradiction must be represented indirectly through logical connectives like $\vdash \bot$ (i.e. "false is true") for contradiction and $\vdash \neg A$ (i.e. "not A is true") or $\vdash A \to \bot$ (i.e. "A implies false is true") for falsehood. A consequence of these indirect encodings is that simplified versions of NJ without a false connective $\bot$ will have trouble speaking about contradictions, and likewise simplifications of NJ without negation $\neg$ will have trouble speaking about falsehoods.

Let's now revisit the basic binary connectives—conjunction ($A \wedge B$), disjunction ($A \vee B$), and implication ($A \supset B$)—by giving their meaning in terms of truth tables that describe the relationship between the truth of a compound proposition and the truth of its parts, as shown in Figure 3.1. Coupled with the interpretation of sequents, this interpretation of connectives gives a simple method of determining the validity of inference rules by checking if the conclusion does indeed follow from the premises. For example, we can validate the inference rules involving conjunction shown in Figure 3.2.

Due to the interaction between entailment in the sequent (separating hypotheses from consequences) and the line of inference (separating premises from conclusions), we have two dimensions for orienting inference rules based on the location of their *primary* proposition (marked with a box in Figure 3.2). On the horizontal axis, rules where the primary proposition appears to the right or left of the turnstyle are called *right* and *left* rules, respectively. On the vertical axis, rules where the primary proposition appears below or above the line of inference are called *introduction* and *elimination* rules, respectively. This gives us four quadrants where the rules of inference for conjunction might live.

- Right introduction: knowing that A is true and B is true is sufficient to conclude that $A \wedge B$ is true.

- Right elimination: known that $A \wedge B$ is true is sufficient to conclude that A is true and likewise that B is true.

- Left introduction: knowing that A is false is sufficient to conclude that $A \wedge B$ is false, and likewise when B is false.

- Left elimination: knowing that $A \wedge B$ is false while both A and B are true is sufficient to deduce a contradiction, as this represents an impossible situation.

Similar inference rules with similar readings can be given for disjunction and implication under the same right/left and introduction/elimination orientations as shown in Figure 3.3 and Figure 3.4.

Notice how the extra judgemental structure provided by sequents allows for simpler versions of some of the particularly complex inference rules from natural deduction in Figure 2.1 that introduce localized assumptions to select premises. In contrast to the NJ inference rule $\supset I$ for right implication introduction which proves $A \to B$ by introducing a local assumption that A is true ($\vdash A$) in the premise which

A	B	$A \wedge B$
False	False	False
False	True	False
True	False	False
True	True	True

A	B	$A \vee B$
False	False	False
False	True	True
True	False	True
True	True	True

A	B	$A \supset B$
False	False	True
False	True	True
True	False	False
True	True	True

FIGURE 3.1. Truth tables for conjunction ($\wedge$), disjunction ($\vee$), and implication ($\supset$).

$$\begin{array}{lcc}
 & \text{Left} & \text{Right} \\[1em]
\text{Elimination} & \dfrac{\boxed{A \wedge B} \vdash \quad \vdash A \quad \vdash B}{\vdash} & \dfrac{\vdash \boxed{A \wedge B}}{\vdash A} \qquad \dfrac{\vdash \boxed{A \wedge B}}{\vdash B} \\[2em]
\text{Introduction} & \dfrac{A \vdash}{\boxed{A \wedge B} \vdash} \qquad \dfrac{B \vdash}{\boxed{A \wedge B} \vdash} & \dfrac{\vdash A \quad \vdash B}{\vdash \boxed{A \wedge B}}
\end{array}$$

FIGURE 3.2. The orientation of deductions for conjunction ($\wedge$).

$$\begin{array}{lcc}
 & \text{Left} & \text{Right} \\[1em]
\text{Elimination} & \dfrac{\boxed{A \vee B} \vdash}{A \vdash} \qquad \dfrac{\boxed{A \vee B} \vdash}{B \vdash} & \dfrac{\vdash \boxed{A \vee B} \quad A \vdash \quad B \vdash}{\vdash} \\[2em]
\text{Introduction} & \dfrac{A \vdash \quad B \vdash}{\boxed{A \vee B} \vdash} & \dfrac{\vdash A}{\vdash \boxed{A \vee B}} \qquad \dfrac{\vdash B}{\vdash \boxed{A \vee B}}
\end{array}$$

FIGURE 3.3. The orientation of deductions for disjunction ($\vee$).

$$\begin{array}{lcc}
 & \text{Left} & \text{Right} \\[1em]
\text{Elimination} & \dfrac{\boxed{A \supset B} \vdash}{\vdash A} \qquad \dfrac{\boxed{A \supset B} \vdash}{B \vdash} & \dfrac{\vdash \boxed{A \supset B} \quad \vdash A}{\vdash B} \\[2em]
\text{Introduction} & \dfrac{\vdash A \quad B \vdash}{\boxed{A \supset B} \vdash} & \dfrac{A \vdash B}{\vdash \boxed{A \supset B}}
\end{array}$$

FIGURE 3.4. The orientation of deductions for implication ($\supset$).

proves B is true ($\vdash B$), the sequent-based right introduction rule in Figure 3.4 instead stores A as a hypothesis in the premise $A \vdash B$ which asserts that A entails B, thereby reducing the implication connective to the implication built into the meaning of the turnstyle. Likewise, In contrast to the NJ inference rule $\vee E$ for right disjunction elimination which introduces local assumptions for both possibilities A and B into two different premises, instead the sequent-based right elimination rule in Figure 3.3 stores the possibilities as hypotheses in the premises $A \vdash$ and $B \vdash$ which assert that A and B are false.

With the dimensions of logical orientation illustrated in Figure 3.2, Figure 3.3, and Figure 3.4, we can identify one of the primary distinctions between natural deduction and the sequent calculus. Natural deduction is exclusively made up of *right rules*—including both right introduction and right elimination—and the sequent calculus is exclusively made up of *introduction rules*—including both right introduction and left introduction.[2] Or in other words, natural deduction is concerned with deducing and using the *truth* of propositions, whereas the sequent calculus is concerned with *introducing* true and false applications of logical connectives. With this fundamental characterization of the sequent calculus in mind, we will delve into Gentzen's LK: the original sequent-based logic.

Gentzen's LK

Gentzen's LK, a simple logic based extensively on the use of sequents to trace local hypotheses and consequences throughout a proof, is given in Figure 3.5. The sequents are built out of (ordered) lists of propositions Γ and Δ, and the inference rules let us build proof trees by stacking inferences on top of one another. We include all the same connectives in LK as we had in NJ: the nullary constants $\top$ and $\bot$, the binary operators $\wedge$, $\vee$, and $\supset$, and quantifiers $\forall$ and $\exists$. Additionally, notice that negation is included as a full-fledged unary connective $\neg A$, whose logical inference rules are easy to define in terms of sequents, instead of encoding it with implication and falsehood as in NJ.

The various inference rules of LK can be thought of in three groups that collectively work toward different objectives. The first group, containing just the axiom (Ax) and cut (Cut) rules, gives the core of LK. The Ax rule lets us draw consequences

[2] But no one, it seems, is interested in left eliminations. A rare exception is the stack calculus (Carraro *et al.*, 2012) which characterizes implication entirely by left rules only.

$$
\begin{aligned}
X, Y, Z \in \mathit{PropVariable} &::= \ldots \\
A, B, C \in \mathit{Proposition} &::= X \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid \neg A \mid A \supset B \mid \forall X.A \mid \exists X.A \\
\Gamma \in \mathit{Hypothesis} &::= A_1, \ldots, A_n \\
\Delta \in \mathit{Consequence} &::= A_1, \ldots, A_n \\
\mathit{Judgement} &::= \Gamma \vdash \Delta
\end{aligned}
$$

Core rules:

$$
\frac{}{A \vdash A}\,\mathit{Ax}
\qquad\qquad
\frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\,\mathit{Cut}
$$

Logical rules:

$$
\frac{}{\Gamma \vdash \top, \Delta}\,\top R
\qquad \text{no } \top L \text{ rule} \qquad \text{no } \bot R \text{ rule} \qquad
\frac{}{\Gamma, \bot \vdash \Delta}\,\bot L
$$

$$
\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\,\wedge R
\qquad
\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge L_1
\qquad
\frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge L_2
$$

$$
\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee R_1
\qquad
\frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee R_2
\qquad
\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\,\vee L
$$

$$
\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,\neg R
\qquad
\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,\neg L
\qquad
\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta}\,{\supset} R
\qquad
\frac{\Gamma \vdash A, \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma', \Gamma, A \supset B \vdash \Delta', \Delta}\,{\supset} L
$$

$$
\frac{\Gamma \vdash A, \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \forall X.A, \Delta}\,\forall R
\qquad
\frac{\Gamma, A\left\{B/X\right\} \vdash \Delta}{\Gamma, \forall X.A \vdash \Delta}\,\forall L
$$

$$
\frac{\Gamma \vdash A\left\{B/X\right\}, \Delta}{\Gamma \vdash \exists X.A, \Delta}\,\exists R
\qquad
\frac{\Gamma, A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma, \exists X.A \vdash \Delta}\,\exists L
$$

Structural rules:

$$
\frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\,\mathit{WR}
\qquad
\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\,\mathit{WL}
\qquad
\frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\,\mathit{CR}
\qquad
\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\,\mathit{CL}
$$

$$
\frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, B, A, \Delta'}\,\mathit{XR}
\qquad
\frac{\Gamma', B, A, \Gamma \vdash \Delta}{\Gamma', A, B, \Gamma \vdash \Delta}\,\mathit{XL}
$$

FIGURE 3.5. The LK sequent calculus for second-order propositional logic: with truth ($\top$), falsehood ($\bot$), conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), implication ($\supset$), and both universal ($\forall$) and existential ($\exists$) propositional quantification.

from hypotheses with the understanding that "A entails A" for any proposition A. The *Cut* rule lets us eliminate intermediate propositions from a proof. For example, the special case of the *Cut* rule where the hypothesis Γ and Γ' and consequences Δ and Δ' are all empty is:

$$\frac{\vdash A \quad A \vdash}{\vdash}\ \mathit{Cut}$$

In other words, if we know that a proposition A is both true ($\vdash A$) and false ($A \vdash$), then we can conclude that a contradiction has taken place ($\vdash$). We can then use the intuitive reading of sequents to extend this reasoning to the general form of *Cut*, meaning that it is valid to allow additional hypotheses and alternate consequences in both premises when eliminating a proposition in this fashion so long as they are all gathered together in the resulting conclusion. If Γ entails *either* A or Δ, and *both* Γ' and A entails Δ', then *both* Γ' and Γ entails *either* Δ' or Δ by cases on which of A or Δ is entailed by Γ: if A is a consequence of Γ, then Δ' is a consequence of the combination of A and Γ', otherwise Δ must be a consequence of Γ.

Both *Ax* and *Cut* play an important part in the overall structure of LK proof trees. The *Ax* serves as the primitive leaves of the proof, signifying that there is nothing interesting to justify because we have just what is needed. The *Cut* lets us use auxiliary proofs or "lemmas" without them appearing in the final conclusion, where on the one hand we show how to derive a proposition A as a consequence and on the other hand we assume A as a hypothesis that may be used in another proof.

The second group of inference rules aims to characterize the logical connectives. These logical rules are generalizations of the introduction rules for the connectives from Figure 3.2, Figure 3.3, and Figure 3.4: the left rules are named with an L and the right rules are named with an R. Compared to the basic inference rules that came from an intuitive understanding of connectives as truth tables, each logical rule is generalized with additional hypotheses and alternative conclusions that are "along for the ride," similar to *Cut*. For example, the two left introduction rules for conjunction in Figure 3.2 are generalized to:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \land B \vdash \Delta}\ \land L_1 \qquad\qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \land B \vdash \Delta}\ \land L_2$$

which say that if Δ is a consequence of A and Γ, then Δ is just as well a consequence of $A \land B$ and Γ (and similarly for B). Since we also consider logical negation $\neg A$ as a connective, it too is equipped with left and right introduction rules in Figure 3.5. These rules have the following special cases when Γ and Δ are empty:

$$\frac{A \vdash}{\vdash \neg A}\,\neg R \qquad\qquad \frac{\vdash A}{\neg A \vdash}\,\neg L$$

In other words, whenever A is false we can infer that $\neg A$ true, and whenever A is true we know $\neg A$ is false. Similarly, the logical rules of the nullary connectives $\top$ and $\bot$ are easy verify by the meaning of sequents. Clearly Γ entails either $\top$ or Δ for any Γ and Δ, since $\top$ is always true, and likewise both Γ and $\bot$ entail Δ because $\bot$ is never true.

The most subtle logical connectives in LK are the quantifiers $\forall$ and $\exists$. The special cases of the introduction rules for $\forall X.A$ and $\exists X.A$ when Γ and Δ are:

$$\frac{\vdash A}{\vdash \forall X.A}\,\forall R \qquad \frac{A\{B/X\} \vdash}{\forall X.A \vdash}\,\forall L \qquad \frac{\vdash A\{B/X\}}{\vdash \exists X.A}\,\exists R \qquad \frac{A \vdash}{\exists X.A \vdash}\,\exists L$$

For universal quantification over the variable X in A, if we can prove that A is true without knowing anything about X then we can infer that $\forall X.A$ is true, and if we can exhibit a counterexample for a specific B such that A with B for X is false then we know the general $\forall X.A$ must be false. Existential quantification over the variable X in A is reversed, so that exhibiting an example for a specific B such that A with B for X is true means that $\exists X.A$ must be true, whereas showing that A is false without knowing anything about X lets us infer that $\exists X.A$ is false.

The extra subtlety of the quantifiers lies in ensuring that we "know nothing else about X." In natural deduction, this fact was expressed as a property of an entire proof sub-tree by checking all the leaves. In the sequent calculus, however, this extra constraint is more easily captured locally as a simple side condition because the "leaves" are all immediately known within the sequents. This side condition states that the variable X does not appear free anywhere else in the sequent, written as the premise $X \notin FV(\Gamma \vdash \Delta)$ in both the $\forall R$ and $\exists L$ rules. Just as in NJ, this extra side condition really is necessary, since without it both quantifiers collapse into one, which is clearly not what we want. In LK, we should expect that a $\forall$ entails the corresponding $\exists$, for example $\forall X.X \vdash \exists X.X$ which is proved as follows:

$$\dfrac{\dfrac{\dfrac{}{Y \vdash Y}\,Ax}{\forall X.X \vdash Y}\,\forall L}{\forall X.X \vdash \exists X.X}\,\exists R$$

But intuitively it shouldn't be that an $\exists$ always entails the corresponding $\forall$. However, consider the following attempted proof of $\exists X.X \vdash \forall X.X$:

$$\dfrac{\dfrac{\dfrac{}{X \vdash X}\,Ax \qquad X \notin FV(\;\vdash X)}{\exists X.X \vdash X}\,\exists L \qquad X \notin FV(\exists X.X \vdash\;)}{\exists X.X \vdash \forall X.X}\,\forall R$$

The only reason that this proof is not valid is because the side conditions on X are not met: $X \notin FV(\exists X.X \vdash\;)$ is true but $X \notin FV(\;\vdash X)$ does not hold. Therefore, the side conditions on the free type variables of sequents in the $\forall R$ and $\exists L$ rules are essential for keeping the intended distinct meanings of the quantifiers.

The third group of inference rules aim to describe the structural properties of the sequents themselves that arise from their meaning. The *weakening* rules say that we can make any proof weaker by adding additional unused hypotheses (WL) or considering alternative unfulfilled consequences (WR) since the presence of irrelevant propositions doesn't matter. The *contraction* rules say that duplicate hypotheses (CL) and duplicate consequences (CR) can just as well be merged into one since redundant repetitions don't matter. And finally, the *exchange* rules say that hypotheses (XL) and consequences (XR) can be swapped since the order of propositions doesn't matter.

Remark 3.1. It may seem strange that the meaning of a sequent with multiple consequences is that only *one* consequence must be true instead of *all* consequences being true. In other words, the consequences of a sequent are *disjunctive* rather than *conjunctive* so that, for example, $A \vdash B, C$ means "A entails B or C" instead of "A entails B and C." One reason for this interpretation is that disjunctive consequences can be weakened but conjunctive consequences cannot. For example, if we already know that "A entails B or C" then we can deduce "A entails B or C or D" for any D because we already know that either B or C is a consequence of A, so the status of D is irrelevant. However, if we already know that "A entails B and C" then we don't know much about "A entails B and C and D" in general, since D might not actually follow from A at all. A similar argument also explains why the hypotheses of a sequent are conjunctive rather than disjunctive. Therefore, the meaning of sequents, where *all* hypotheses must entail *one* consequence, is essential for enabling weakening on both sides of entailment. *End remark* 3.1.

Example 3.1. Through the exclusive use of introduction rules for treating logical connectives, LK enables a "bottom up" style of building proofs by starting with a final sequent as a goal that we would like to prove and building the rest of the proof up from there. When read in reverse, each logical rule identifies a connective in the goal below the line of inference and breaks it down into simpler sub-goals above the line. For example, let's revisit Example 2.1 and consider how to build an LK proof that the proposition $((A \land B) \land C) \supset (B \land A)$ is true. As in NJ, we begin with the sequent $\vdash ((A \land B) \land C) \supset (B \land A)$ as the goal and notice that the primary connective exposed in the only proposition available is implication, so we can apply the right implication rule:

$$\dfrac{\begin{array}{c}\vdots\\ (A \land B) \land C \vdash B \land A\end{array}}{\vdash ((A \land B) \land C) \supset (B \land A)}\,{\supset}R$$

Next, we may break down the conjunction in the consequence $B \land A$ with the right conjunction rule, splitting the proof into two parts:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ (A \land B) \land C \vdash B\end{array} \quad \begin{array}{c}\vdots\\ (A \land B) \land C \vdash A\end{array}}{(A \land B) \land C \vdash B \land A}\,{\land}R}{\vdash ((A \land B) \land C) \supset (B \land A)}\,{\supset}R$$

At this point, the consequences of both our goals are generic, lacking any specific connectives to work with, which is where the proof differs from the proof in Example 2.1. Instead of moving to build the proof top-down as in NJ, in LK we shift our attention to the left and begin breaking down the hypotheses. Since the hypothesis $(A \land B) \land C$ contains a superfluous C, we use the first left conjunction rule in both branches of the proof to discard it:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ A \land B \vdash B\end{array}}{(A \land B) \land C \vdash B}\,{\land}L_1 \quad \dfrac{\begin{array}{c}\vdots\\ A \land B \vdash A\end{array}}{(A \land B) \land C \vdash A}\,{\land}L_1}{(A \land B) \land C \vdash B \land A}\,{\land}R}{\vdash ((A \land B) \land C) \supset (B \land A)}\,{\supset}R$$

Now we may apply another left conjunction rule to select the appropriate hypothesis needed for both sub-proofs:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ B \vdash B\end{array}}{A \land B \vdash B}\,\land L_2}{(A \land B) \land C \vdash B}\,\land L_1 \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ A \vdash A\end{array}}{A \land B \vdash A}\,\land L_1}{(A \land B) \land C \vdash A}\,\land L_1}{(A \land B) \land C \vdash B \land A}\,\land R}{\vdash ((A \land B) \land C) \supset (B \land A)}\,\supset R$$

And finally, we can now close off both sub-proofs with the *Ax* rule, finishing the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\,Ax}{A \land B \vdash B}\,\land L_2}{(A \land B) \land C \vdash B}\,\land L_1 \qquad \dfrac{\dfrac{\dfrac{}{A \vdash A}\,Ax}{A \land B \vdash A}\,\land L_1}{(A \land B) \land C \vdash A}\,\land L_1}{(A \land B) \land C \vdash B \land A}\,\land R}{\vdash ((A \land B) \land C) \supset (B \land A)}\,\supset R$$

End example 3.1.

Remark 3.2. The traditional LK sequent calculus from Figure 3.5 presents the structural properties of sequents—exchange, weakening, and contraction—explicitly in the form of inference rules. However, there are alternate sequent calculi and variations on LK that forgo these structural rules by baking the properties deeper into the logic itself. The first change along this line is to treat the hypotheses and consequences of sequents as *unordered* collections of propositions, for example building sequents out of sets or multisets. This way, the exchange rules *XL* and *XR* don't do anything at all, since the sequents in the premise and conclusion are considered identical. The second change is to rephrase the core axiom and cut rules in a way that bakes in weakening and contraction as follows:

$$\dfrac{}{\Gamma, A \vdash A, \Delta}\,Ax \qquad\qquad \dfrac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\,Cut$$

Contraction can be derived from these new *Ax* and *Cut* rules. *CL* is derived as:

$$\dfrac{\Gamma, A, A \vdash \Delta \quad \dfrac{}{\Gamma, A \vdash A, \Delta}\,Ax}{\Gamma, A \vdash \Delta}\,Cut$$

and the derivation of *CR* is similar. Weakening, unfortunately, cannot be directly derived in the same manner as contraction, but instead it is *admissible*. That is to say, given any proof of the sequent $\Gamma \vdash \Delta$, we can build similar proofs $\Gamma, A \vdash \Delta$ and $\Gamma \vdash A, \Delta$ by pushing the unused A through the proof until it is finally discarded by the generalized *Ax* rule.

In terms of provability—the question of which sequents can conclude a valid proof tree—the versions of LK with explicit and implicit structural rules are the same. In the implicit system, exchange is invisible, contraction is a consequence of axiom and cut, and all weakening is pushed to the leaves. Furthermore, the two different versions of the axiom and cut rules are interderivable with respect to their different logics. The explicit *Ax* rule in Figure 3.5 is a special case of the implicit one above, whereas the implicit *Ax* rule can be expanded into many weakenings followed by the explicit rule. Likewise, the explicit *Cut* rule can be derived from the implicit rule by weakening the two premises until they match, whereas the implicit *Cut* rule can be derived from the explicit rule by contracting the result of the conclusion to remove the duplication. Therefore, up to provability, the choice between these two different styles for handling the structural properties of sequents are a matter of taste. On the same subject, it's also sensible to consider an alternate version of left implication introduction that duplicates rather than splitting hypotheses and consequences among the premises in the style of our revised *Cut* above:

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \supset B \vdash \Delta} \supset\! L$$

In the presence of structural properties (either explicit or implicit), these two $\supset\! L$ rules are equivalent up to provability. However, if we want a more refined view of the structural properties, as in sub-structural logics like linear logic (Girard, 1987), then these differences become more acute and must be considered carefully. *End remark* 3.2.

Consistency and cut elimination

One of Gentzen's motivations for developing the LK sequent calculus was to study the consistency of natural deduction. A consistent logic does not prove a contradiction, so that no proposition is proven both true and false. More specifically, we can say that a sequent calculus is *consistent* whenever there is no proof of the empty sequent $\vdash$.

For a logic like LK, these two conditions are the same: from a contradiction weakening gives us $\vdash A$ and $A \vdash$ for any A, and from any A, that's proven both true and false, *Cut* gives us $\vdash$. Consistency of logics like LK is important because without consistency provability is meaningless: it's not particularly interesting to exhibit a proof that some proposition A is true when we already know of a single proof that shows every proposition is true (and false)!

So in the interest of showing LK's consistency, how we might possibly begin to build a proof of the empty sequent from the bottom up? Let's consider which of LK's inference rules (from Figure 3.5) could possibly deduce $\vdash$. It can't be any of the structural rules because they all force at least one hypothesis or consequence in the conclusion below the line. Likewise, it can't be any of the logical rules: since they are introduction rules, they all include at least one proposition built from a connective on either side of the deduced sequent. It also can't be the axiom rule, which only deduces simple non-empty sequents of the form $A \vdash A$. Indeed, the *only* inference rule that might ever deduce an empty sequent—and therefore lead to inconsistency—is *Cut* as shown previously.

This observation that only cuts can lead to contradictions is Gentzen's (1935b) great insight to logical consistency. If we want to know that a sequent calculus like LK is consistent, it's enough to ask if the *Cut* rule is important for provability. If *Cut* is not essential in any proof, so any provable sequent can be deduced without the help of *Cut*, then $\vdash$ is unprovable since it cannot be deduced without *Cut*. This application highlights the importance of Gentzen's (1935a) *cut elimination* (originally called *Hauptsatz*), and its phrasing in the sequent calculus, which says that every LK proof can be reduced to a cut-free one.

Theorem 3.1 (Cut elimination)**.** *For all LK proofs of* $\Gamma \vdash \Delta$*, there exists an alternate LK proof of* $\Gamma \vdash \Delta$ *that does not contain any use of the Cut rule.*

The proof of cut elimination can be divided into two main parts: the *logical* steps and the *structural* steps. The logical steps of cut elimination consider the cases when we have a cut between two proof trees ending in the left and right rules for the same connective occurring in the same proposition, and show how to rewrite the proof into a new one that does not mention that particular connective. The structural steps of cut elimination handle all the other cases where we do not have a left and right introduction for the same proposition facing one another in a cut. These steps involve rewriting the structure of the proof and propagating the rules until the relevant logical steps can take over. The final ingredient is to ensure that this procedure for eliminating cuts always gives a definite result, and does not spin off into an infinite regress.

Example 3.2. Notice how different inference rules of LK treat the division of extraneous hypotheses and consequences among multiple premises differently. On the one hand, rules like $\wedge R$ and $\vee L$ duplicate the side propositions Γ and Δ from the conclusion to both premises. On the other hand, rules like *Cut* and $\supset L$ merge different side propositions from the two premises into the common conclusion, creating an ordering between them during the merge. Why are these particular rules given in such different styles, and why is the particular merge order chosen? One way to understand the impact of these details is to look at the interaction between the logical and structural rules during cut elimination, so let's examine a few exemplar steps of the cut elimination procedure.

The first, and the most trivial, case is when we cut an axiom with an existing proof $\mathcal{D}$ of $\Gamma \vdash A, \Delta$ or $\mathcal{E}$ of $\Gamma, A \vdash \Delta$. This particular maneuver doesn't add anything interesting to the nature of the existing proof, and so correspondingly eliminating the cut should just give the same proof back unchanged, as we can see in both cases:

$$\dfrac{\begin{matrix}\mathcal{D}\\ \vdots\\ \Gamma \vdash A, \Delta\end{matrix} \quad \dfrac{}{A \vdash A}\,\mathit{Ax}}{\Gamma \vdash A, \Delta}\,\mathit{Cut} \Longrightarrow \begin{matrix}\mathcal{D}\\ \vdots\\ \Gamma \vdash A, \Delta\end{matrix} \qquad \dfrac{\dfrac{}{A \vdash A}\,\mathit{Ax} \quad \begin{matrix}\mathcal{E}\\ \vdots\\ \Gamma, A \vdash \Delta\end{matrix}}{\Gamma, A \vdash \Delta}\,\mathit{Cut} \Longrightarrow \begin{matrix}\mathcal{E}\\ \vdots\\ \Gamma, A \vdash \Delta\end{matrix}$$

Notice here that cutting an axiom with both $\mathcal{D}$ and $\mathcal{E}$ does not change the sequent in either conclusion, which comes from the precise way that *Cut* merges the side propositions in the two premises. For $\mathcal{D}$, the extra consequence A coming from the axiom $A \vdash A$ replaces the cut A in exactly the right position, and likewise for $\mathcal{E}$. If *Cut* put the propositions of its conclusion in any other order, then we would need to exchange the result of one or both of the above steps with *XL* and *XR* to put them back into the right order.

Moving on to a logical step, consider what happens when compatible $\wedge R$ and $\wedge L_1$ introductions, with premises $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{E}$ respectively, meet in a *Cut*:

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}_1\\ \vdots\\ \Gamma \vdash A, \Delta\end{matrix} \quad \begin{matrix}\mathcal{D}_2\\ \vdots\\ \Gamma \vdash B, \Delta\end{matrix}}{\Gamma \vdash A \wedge B, \Delta}\,\wedge R \quad \dfrac{\begin{matrix}\mathcal{E}\\ \vdots\\ \Gamma', A \vdash \Delta'\end{matrix}}{\Gamma', A \wedge B \vdash \Delta'}\,\wedge L_1}{\Gamma', \Gamma \vdash \Delta', \Delta}\,\mathit{Cut} \Longrightarrow \dfrac{\begin{matrix}\mathcal{D}_1\\ \vdots\\ \Gamma \vdash A, \Delta\end{matrix} \quad \begin{matrix}\mathcal{E}\\ \vdots\\ \Gamma', A \vdash \Delta'\end{matrix}}{\Gamma', \Gamma \vdash \Delta', \Delta}\,\mathit{Cut}$$

Reducing this cut involves selecting the appropriate premise $\mathcal{D}_1$ of the $\wedge R$ introduction so that it can meet with the single premise of $\wedge L_1$. The number of cuts are not reduced by this step, but instead the primary proposition $A \wedge B$ of the cut has been reduced to A, which (non-trivially) justifies why this step is making progress in the cut elimination procedure.

Not every cut-elimination step winds up so neatly organized, unfortunately, and sometimes the result is necessarily out of order and must be corrected. For example, consider the following reduction step of a *Cut* between compatible $\neg R$ and $\neg L$ inferences with premises $\mathcal{D}$ and $\mathcal{E}$ respectively:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma, A \vdash \Delta\end{array}}{\Gamma \vdash \neg A, \Delta}\neg R \quad \dfrac{\begin{array}{c}\mathcal{E}\\ \vdots\\ \Gamma' \vdash A, \Delta'\end{array}}{\Gamma', \neg A \vdash \Delta'}\neg L}{\Gamma', \Gamma \vdash \Delta', \Delta}\mathit{Cut} \implies \dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ \vdots\\ \Gamma' \vdash A, \Delta'\end{array} \quad \begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma, A \vdash \Delta\end{array}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\mathit{Cut}}{\Gamma', \Gamma \vdash \Delta', \Delta}\mathit{XL}, \mathit{XR}$$

Here, the *Cut* we get from reducing the proposition $\neg A$ to A results in a sequent that is out of order compared to the conclusion we started with. Thus, we need to re-order the sequent with some number of *XL* and *XR* exchanges to restore the original conclusion. The fact that reducing a negation introduction cut inverts the order of propositions comes from the inherent inversion of negation: there's no obvious way to prevent this scenario by modifying *Cut*.

A similar re-ordering occurs with implication, where a *Cut* between compatible $\supset R$ and $\supset L$ inferences, with premises $\mathcal{D}$, $\mathcal{E}_1$, and $\mathcal{E}_2$, can be reduced as follows:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma, A \vdash B, \Delta\end{array}}{\Gamma \vdash A \supset B, \Delta}{\supset} R \quad \dfrac{\begin{array}{c}\mathcal{E}_1\\ \vdots\\ \Gamma' \vdash A, \Delta'\end{array} \quad \begin{array}{c}\mathcal{E}_2\\ \vdots\\ \Gamma'', B \vdash \Delta''\end{array}}{\Gamma'', \Gamma', A \supset B \vdash \Delta'', \Delta'}{\supset} L}{\Gamma'', \Gamma', \Gamma \vdash \Delta'', \Delta', \Delta}\mathit{Cut} \implies \dfrac{\dfrac{\begin{array}{c}\mathcal{E}_1\\ \vdots\\ \Gamma' \vdash A, \Delta'\end{array} \quad \dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma, A \vdash B, \Delta\end{array} \quad \begin{array}{c}\mathcal{E}_2\\ \vdots\\ \Gamma'', B \vdash \Delta''\end{array}}{\Gamma'', \Gamma, A \vdash \Delta'', \Delta}\mathit{Cut}}{\Gamma'', \Gamma, \Gamma' \vdash \Delta'', \Delta, \Delta'}\mathit{Cut}}{\Gamma'', \Gamma', \Gamma \vdash \Delta'', \Delta', \Delta}\mathit{XL}, \mathit{XR}$$

Here, we start with the side-propositions of $\mathcal{E}_1$ and $\mathcal{E}_2$ merged together with $\supset L$, but after reducing the *Cut*, $\mathcal{D}$ cuts in between the two of them, so the final sequent must be re-ordered to match the original conclusion. The need to place $\mathcal{D}$ in the middle comes from the fact that its concluding sequent has A on the left and B on the right, so our only available cuts must correspondingly place $\mathcal{E}_1$ to the left and $\mathcal{E}_2$ to the right, no matter how they are nested.

Finally, we can see how the free variable side conditions on the $\forall R$ and $\exists L$ rules play a key role in cut elimination. For example, consider the following reduction step of a cut between compatible $\forall R$ and $\forall L$ inferences with $\mathcal{D}$ and $\mathcal{E}$ respectively:

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ \vdots\\ \Gamma \vdash A, \Delta\end{matrix}}{\Gamma \vdash \forall X.A, \Delta}\forall R \quad \dfrac{\begin{matrix}\mathcal{E}\\ \vdots\\ \Gamma', A\{B/X\} \vdash \Delta\end{matrix}}{\Gamma', \forall X.A \vdash \Delta'}\forall L}{\Gamma', \Gamma \vdash \Delta', \Delta}\mathit{Cut} \Longrightarrow \dfrac{\begin{matrix}\mathcal{D}\{B/X\}\\ \vdots\\ \Gamma \vdash A\{B/X\}, \Delta\end{matrix} \quad \begin{matrix}\mathcal{E}\\ \vdots\\ \Gamma', A\{B/X\} \vdash \Delta\end{matrix}}{\Gamma', \Gamma \vdash \Delta', \Delta}\mathit{Cut}$$

Notice that in order to make a direct cut between $\mathcal{D}$ and $\mathcal{E}$, we need to substitute B for X in $\mathcal{D}$ to make the two sides match up properly. The fact that X does not occur free in $\Gamma \vdash \Delta$ means that after substitution, both Γ and Δ remain unchanged in the conclusion of the proof. If instead X appeared free somewhere in Γ or Δ, then the logical cut elimination step for $\forall$ would change the conclusion which ruins the result of the procedure. *End example* 3.2.

Remark 3.3. The side conditions on the $\forall R$ and $\exists L$ rules are not just a useful aid to cut elimination, but are crucial to the entire endeavor. More specifically, if we removed the side condition from these two inference rules, then LK is inconsistent because we can directly derive a contradiction; and since cut elimination implies that contradictions cannot be derived, it therefore becomes impossible. One such contradiction is built in three parts, and is similar to the faulty NJ proof of false in Remark 2.2. First, we can prove that $\exists X.X$ is true because there is *some* provably true proposition in LK, for example $Y \supset Y$ or just $\top$. Second, we can prove that $\forall X.X$ is false because there is *some* provably false proposition in LK, for example $(\neg Y) \wedge Y$ or just $\bot$. Third, recall that without the side conditions on free propositional variables, we can derive a proof of $\exists X.X \vdash \forall X.X$, which is the glue that connects the first two parts together via cuts. In total, we would be able to derive the following contradiction in LK:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \top}\top R}{\vdash \exists X.X}\exists R \quad \dfrac{\dfrac{\dfrac{}{X \vdash X}Ax \quad X \notin FV(\vdash X)}{\exists X.X \vdash X}\exists L \quad X \notin FV(\exists X.X \vdash\ )}{\exists X.X \vdash \forall X.X}\forall R}{\vdash \forall X.X}\mathit{Cut} \quad \dfrac{\dfrac{}{\bot \vdash}\bot L}{\forall X.X \vdash}\forall L}{\vdash}\mathit{Cut}$$

which is only ruled out by the side conditions on $\forall R$ and $\exists L$ that prevent a proof of the sequent $\exists X.X \vdash \forall X.X$. In this particular proof, the side condition $X \notin FV(\exists X.X \vdash)$ is satisfied because X is bound in $\exists X.X$ so X is indeed not free in $\exists X.X \vdash$, but the side condition $X \notin FV(\vdash X)$ is clearly violated. The other possible proof which switches the order of the $\exists L$ and $\forall R$ rules similarly violates the side condition $X \notin FV(X \vdash)$ forced by $\forall R$. *End remark* 3.3.

Logical duality

Another application of sequent calculi is to study the *dualities* of logic through the deep symmetries of the system (Gentzen, 1935b). The turnstyle of entailment ($\vdash$) provides the pivot of duality separating left from right and true from false. Logical duality in the LK sequent calculus expresses a relationship between the connectives that follows De Morgan's laws about the way negation distributes over conjunction and disjunction:

$$\neg(A \wedge B) \dashv\vdash (\neg A) \vee (\neg B)$$
$$\neg(A \vee B) \dashv\vdash (\neg A) \wedge (\neg B)$$

Where we interpret the equivalence relation $A \dashv\vdash B$ as the mutual provability of A and B: that both $A \vdash B$ and $B \vdash A$ are provable. Focusing on the opposite roles of the left and right sides of a sequent, we can immediately observe that the introduction rules of conjunction and disjunction from Figure 3.5 are mirror images of one another by flipping the sequents across their turnstyle. Similarly, both the $\forall$ and $\exists$ are duals to one another, and negation is its own dual, with both $\neg R$ and $\neg L$ reflecting the same inference flipped about entailment.

But what about implication? After examining Figure 3.5, there doesn't seem to be any logical connective that serves as implication's dual counterpart. Fortunately, the symmetric nature of sequents lets us discover the dual of implication by just syntactically flipping the $\supset R$ and $\supset L$ inferences, giving us the following inferences rules for a new connective $B - A$:

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash B - A, \Delta, \Delta'} -R \qquad\qquad \frac{\Gamma, B \vdash A, \Delta}{\Gamma, B - A \vdash \Delta} -L$$

But what does this new connective, the dual of implication, mean? By excluding all side hypotheses and consequences so that $\Gamma, \Gamma', \Delta, \Delta'$ are all empty in the style of Figure 3.4, we can read off the basic truth and falsehood facts from the above rules. On the one hand, the $-R$ rule says that $B - A$ is true whenever B is true and A is false. On the other hand, the $-L$ rule says that $B - A$ must be false whenever B entails A. Therefore, the proposition $B - A$ can be thought of as the *subtraction* of A from B or equivalently the *complement* of A with respect to B, so that $B - A$ can be read as "B but not A."

Remark 3.4. Another method for discovering the implication's dual is by reducing these two rather complex connectives into simpler forms. Notice that, since LK is a classical logic, implication is equivalent to an encoding based on disjunction and negation, up to provability:

$$A \supset B \dashv\vdash (\neg A) \vee B$$

since A implies B is true if and only if either B is true or A is false. The proofs justifying this encoding in LK are:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,Ax}{A, (\neg A) \vdash}\,\neg L \qquad \dfrac{}{B \vdash B}\,Ax}{A, (\neg A) \vee B \vdash B}\,\vee L}{(\neg A) \vee B, A \vdash B}\,XL}{(\neg A) \vee B \vdash A \supset B}\,\supset R
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,Ax}{\vdash \neg A, A}\,\neg R}{\vdash (\neg A) \vee B, A}\,\vee R_1}{\vdash A, (\neg A) \vee B}\,XR \qquad \dfrac{\dfrac{}{B \vdash B}\,Ax}{B \vdash (\neg A) \vee B}\,\vee R_2}{A \supset B \vdash (\neg A) \vee B, (\neg A) \vee B}\,\supset L}{A \supset B \vdash (\neg A) \vee B}\,CR$$

We also have an encoding of subtraction in terms of conjunction and negation:

$$B - A \dashv\vdash B \wedge (\neg A)$$

which is provable similarly to the encoding of implication. We can now use the above encodings to calculate the negation of implication with De Morgan's laws, using the fact that conjunction is provably commutative—$A \wedge B \dashv\vdash B \wedge A$ for any A and B:

$$\begin{aligned}
\neg(A \supset B) &\dashv\vdash \neg((\neg A) \vee B) \\
&\dashv\vdash (\neg(\neg A)) \wedge (\neg B) \\
&\dashv\vdash (\neg B) \wedge (\neg(\neg A)) \\
&\dashv\vdash (\neg B) - (\neg A)
\end{aligned}$$

Duality of sequents:

$$(\Gamma \vdash \Delta)^\perp \triangleq \Delta^\perp \vdash \Gamma^\perp \qquad\qquad (A_1, \dots, A_n)^\perp \triangleq A_n^\perp, \dots, A_1^\perp$$

Duality of propositions:

$$\begin{aligned}
(X)^\perp &\triangleq X & (\neg A)^\perp &\triangleq \neg(A^\perp) \\
\top^\perp &\triangleq \bot & \bot^\perp &\triangleq \top \\
(A \wedge B)^\perp &\triangleq (A^\perp) \vee (B^\perp) & (A \vee B)^\perp &\triangleq (A^\perp) \wedge (B^\perp) \\
(A \supset B)^\perp &\triangleq (B^\perp) - (A^\perp) & (B - A)^\perp &\triangleq (A^\perp) \supset (B^\perp) \\
(\forall X.A)^\perp &\triangleq \exists X.(A^\perp) & (\exists X.A)^\perp &\triangleq \forall X.(A^\perp)
\end{aligned}$$

FIGURE 3.6. Duality in the LK sequent calculus.

The dual is then recovered from the fact that $A^\perp \dashv\vdash (\neg A)^*$, where A^* stands for A with all propositional variables X replaced with $\neg X$. Therefore, we can also derive the dual of implication by encoding it and its dual with conjunction, disjunction, and negation. *End remark* 3.4.

With the dual of implication at hand, we can properly express the duality of sequent calculus proofs—for every LK proof $\mathcal{D}$ of a sequent:

$$\begin{array}{c} \mathcal{D} \\ \vdots \\ A_n, \dots, A_2, A_1 \vdash B_1, B_2, \dots, B_m \end{array}$$

there is a dual proof $\mathcal{D}^\perp$ of the dual sequent:

$$\begin{array}{c} \mathcal{D}^\perp \\ \vdots \\ B_m^\perp, \dots, B_2^\perp, B_1^\perp \vdash A_1^\perp, A_2^\perp, \dots, A_n^\perp \end{array}$$

The duality relation on judgements and propositions, is given in Figure 3.6. Note that the duality operation $A^\perp$ may be understood as taking the negation of the proposition, $\neg A$, and pushing the negation inward all the way using the De Morgan laws, until an unknown proposition variable X is reached (Gentzen, 1935b).[3]

[3] Note that Gentzen did not consider the dual counterpart to implication as a connective, as we do, but rather eliminated implication from the system by encoding it in terms of disjunction and negation given above for the purposes of establishing duality.

Theorem 3.2 (Logical duality). *For any LK proof $\mathcal{D}$ of the sequent $\Gamma \vdash \Delta$, there exists a dual proof $\mathcal{D}^\perp$ of the dual sequent $\Delta^\perp \vdash \Gamma^\perp$.*

Due to the natural syntactic symmetry of the LK sequent calculus, logical duality comes from an exchange between left and right: left rules mirror right rules and hypotheses to the left of entailment mirror consequences to the right. Thus, establishing logical duality in the sequent calculus follows from a straightforward induction on the structure of proofs, working from the bottom conclusion up to the axioms.

Example 3.3. To illustrate how the left and right sides of proofs get swapped, consider the case when the bottom conclusion is inferred from a use of the $\wedge R$ rule:

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma \vdash A, \Delta\end{array} \quad \begin{array}{c}\mathcal{E}\\ \vdots\\ \Gamma \vdash B, \Delta\end{array}}{\Gamma \vdash A \wedge B, \Delta}\ \wedge R$$

Then by the inductive hypothesis, we get a proof $\mathcal{D}^\perp$ of $(\Gamma \vdash A, \Delta)^\perp \triangleq \Delta^\perp, A^\perp \vdash \Gamma^\perp$ and a proof $\mathcal{E}^\perp$ of $(\Gamma \vdash B, \Delta)^\perp \triangleq \Delta^\perp, B^\perp \vdash \Gamma^\perp$, from which we can deduce $(\Gamma \vdash A \wedge B, \Delta)^\perp \triangleq \Delta^\perp, (A^\perp) \vee (B^\perp) \vdash \Gamma^\perp$ by $\vee L$:

$$\dfrac{\begin{array}{c}\mathcal{D}^\perp\\ \vdots\\ \Delta^\perp, A^\perp \vdash \Gamma^\perp\end{array} \quad \begin{array}{c}\mathcal{E}^\perp\\ \vdots\\ \Delta^\perp, B^\perp \vdash \Gamma^\perp\end{array}}{\Delta^\perp, A^\perp \vee B^\perp \vdash \Gamma^\perp}\ \vee L$$

End example 3.3.

Remark 3.5. The duality of proofs in the LK sequent calculus means that if a proposition A is true, so that we have a proof of $\vdash A$, then its dual must be false, so that we have a proof of $A^\perp \vdash$. Analogously, if a proposition A is false, then its dual must be true. For example, consider the following general proof that the contradictory proposition $A \wedge (\neg A)$ is false:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax}{A \wedge (\neg A) \vdash A}\ \wedge L_1}{A \wedge (\neg A), \neg A \vdash}\ \neg L}{A \wedge (\neg A), A \wedge (\neg A) \vdash}\ \wedge L_2}{A \wedge (\neg A) \vdash}\ CL$$

For free, duality gives us a general proof that the law of excluded middle, $A \lor (\neg A)$, is true:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,Ax}{A \vdash A \lor (\neg A)}\,\lor R_1}{\vdash \neg A, A \lor (\neg A)}\,\neg R}{\vdash A \lor (\neg A), A \lor (\neg A)}\,\lor R_2}{\vdash A \lor (\neg A)}\,CR$$

This is not a trivial property—the fact that the LK sequent calculus can prove the law of excluded middle means that it is a proof system for *classical logic*. In contrast, *intuitionistic logic* is missing duality since it accepts non-contradiction, $\neg(A \land (\neg A))$, in general but rejects the universal truth of laws like excluded middle or double negation elimination $((\neg(\neg A)) \supset A)$, only allowing for specialized proofs depending on the particular proposition A in question. Intuitionistic logic also only validates three of the four aforementioned De Morgan laws, rejecting $\neg(A \land B) \vdash (\neg A) \lor (\neg B)$ in particular, showing another break of duality. Gentzen's (1935a) system NJ of natural deduction is naturally a proof system for intuitionistic logic, in contrast with the LK sequent calculus which is classical.

However, notice that the LK proof of excluded middle made critical use of multiple consequences and contraction on the right of the sequent in order to apply both $\lor R_2$ and $\lor R_1$ to the same original consequence. Without the ability to manipulate sequents with multiple consequences, the proof that $A \lor (\neg A)$ is true would not be possible. Indeed, such a restriction would break the symmetry of LK—as multiple hypotheses cannot be mirrored into multiple consequences—and destroy the duality that let us convert the law of non-contradiction into law of excluded middle. As it turns out, Gentzen (1935a) also introduced a sequent calculus called LJ as a restriction of LK where sequents could only ever contain one consequence, which is instead a sequent calculus system for intuitionistic logic of equal provability strength as NJ. Note that with this restriction, LJ effectively removes the right structural rules *WR*, *CR*, and *XR* since they involve sequents with more than one consequence. From the other perspective, generalizing natural deduction with multiple consequences turns it into a proof system for classical logic (Parigot, 1992; Ariola & Herbelin, 2003). Therefore, we can summarize that the difference between a single-consequence and multiple-consequence proof systems can mean the difference between intuitionistic and classical logic. *End remark* 3.5.

The Core Calculus

Today, the Curry-Howard isomorphism (Curry *et al.*, 1958; Howard, 1980; de Bruijn, 1968) is a far-reaching thesis that each logic corresponds to a foundational programming language: the propositions of logic can be seen as types of programs and the proofs of those propositions can be seen as programs themselves. The shining example of this recurring correspondence is between Gentzen's (1935a) natural deduction and Church's (1932) λ-calculus. However, the logics of natural deduction and the sequent calculus are rather different from one another. As previously discussed, one major point of distinction between the two styles of logic is that natural deduction is right-handed, favoring exclusively right rules for logical connectives, whereas the sequent calculus is ambidextrous, favoring introduction rules on both the left and right sides of entailment. That means that the sequent calculus does not correspond to the λ-calculus the same way that natural deduction does. So what might a programming language based on a sequent calculus like LK look like?

From natural deduction's right-handed nature, we get an expression-oriented language like the λ-calculus: all the phrases of the language work toward producing some result corresponding to the primary consequence on the right, and so they may all be (potentially) composed together. But the sequent calculus is ambidextrous, containing both left- and right-handed rules, and regularly deals with sequents like $A, \neg A \vdash$ that lack any particular consequence to speak of. Without a consequence, how can we say what type of result to expect from a program corresponding to the sequent $A, \neg A \vdash$, or that it even produces a result at all? More generally, notice that we can classify the rules of LK from Figure 3.5 by the three different kinds of sequents they can deduce: those with a primary consequence of interest like in the right rules, those with a primary hypothesis of interest like in the left rules, and those with no particular proposition of interest (including possibly the empty sequent) like in the cut rule. If we interpret LK as a programming language, it seems reasonable that each of these different kinds of sequents correspond to a different basic kind of phrase in the language, whose composition is guided by the forms of the inference rules.

Before delving into the entirety of LK, let's first consider a *core* language shown in Figure 3.7, Herbelin's (2005) $\mu\tilde{\mu}$-calculus, that corresponds to the core part of LK and lies in the heart of every sequent-based language we will explore. Notice that the language of types in this core lacks any logical connectives, so that the only types are uninterpreted variables X, Y, Z, *etc.* The $\mu\tilde{\mu}$-calculus is a bare language for describing

$$\begin{array}{rlrl}
A, B, C \in \mathit{Type} ::= X & & X, Y, Z \in \mathit{TypeVariable} ::= \ldots \\
c \in \mathit{Command} ::= \langle v \| e \rangle & & & \\
v \in \mathit{Term} ::= x \mid \mu\alpha.c & & x, y, z \in \mathit{Variable} ::= \ldots \\
e \in \mathit{CoTerm} ::= \alpha \mid \tilde{\mu}x.c & & \alpha, \beta, \gamma \in \mathit{CoVariable} ::= \ldots \\
\Gamma \in \mathit{InputEnv} ::= x_1 : A_1, \ldots, x_n : A_n & & \Delta \in \mathit{OutputEnv} ::= \alpha_1 : A_1, \ldots, \alpha_n : A_n
\end{array}$$

$$\mathit{Judgement} ::= c : (\Gamma \vdash \Delta) \mid (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta)$$

Core rules:

$$\frac{}{x : A \vdash x : A \mid}\ \mathit{VR} \qquad \frac{}{\mid \alpha : A \vdash \alpha : A}\ \mathit{VL}$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ \mathit{AR} \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\ \mathit{AL}$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : A \vdash \Delta'}{\langle v \| e \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)}\ \mathit{Cut}$$

FIGURE 3.7. $\mu\tilde{\mu}$: The core language of the sequent calculus.

only input, output, and interactions: the types on the right side of a sequent describe the outputs of a program and the types on the left side of a sequent describe the inputs of a program. When the two opposite sides come together—when the opposed forces of input and output meet—we have an interaction that sparks computation. Note that the type system brings out an aspect of deduction that was implicit in the sequent calculus: the role of a distinguished *active* proposition that is currently under consideration. For example, in the $\wedge R$ rule from Figure 3.5, we are currently trying to prove the proposition $A \wedge B$, so it is considered the active proposition of the sequent $\Gamma \vdash A \wedge B, \Delta$.

By putting attention on (at most one) active proposition, we get three classifications of sequents: active on the right, active on the left, or passive (without an active proposition on either side). These three forms of sequents likewise classify three different forms of $\mu\tilde{\mu}$ expressions that might be part of a program:

- An active sequent on the right ($\Gamma \vdash v : A \mid \Delta$) describes a *term* v that sends information of type A as its output (that is, v is a *producer* of type A).

- An active sequent on the left ($\Gamma \mid e : A \vdash \Delta$) describes a *co-term* e that receives information of type A as its input (that is, e is a *consumer* of type A).

- A passive sequent $(c : (\Gamma \vdash \Delta))$ describes a *command* c that is an executable program capable of running on its own without any distinguished input or output.

In each case, the environments Γ and Δ describe any additional inputs and outputs to an expression by specifying the type of *free variables* $(x, \dots)$ and *free co-variables* $(\alpha, \dots)$ that expression might reference, respectively.

The expressions of the $\mu\tilde{\mu}$-calculus come from the axiom and cut rules of LK plus an additional pair of *activation* rules AR and AL. The Ax rule of LK is divided into two separate rules in $\mu\tilde{\mu}$: the VR rule creates a term by just referring to a variable available from its environment, and similarly the VL rule creates a co-term by referring to a co-variable. The Cut rule connects a term and co-term that are waiting to send and receive information of the same type, so that the output of the term is forwarded to the co-term as input (and dually, the input of the co-term is drawn from the output of the term). Finally, the *activation* rules AR and AL pick a particular (co-)variable from the environment of a command to activate by creating an output or input abstraction, respectively. Intuitively, if the variable x stands for an unknown input in a command c, then the input abstraction $\tilde{\mu}x.c$ is a co-term that, when given a place to draw information, will bind that location to the input channel x while running c. Dually, if the co-variable α stands for an unknown output in a command c, then the output abstraction $\mu\alpha.c$ is a term that, when given a place to send information, will bind that location to the output channel α while running c.

Having examined the *static* properties of the $\mu\tilde{\mu}$-calculus—its syntax and types—we still need to consider the *dynamic* properties of $\mu\tilde{\mu}$, to explain what it means to run a program. To say "what is computation in the sequent calculus?" we turn to cut elimination (previously mentioned in Section 3.1) which outlines a method of reducing commands as the main unit of computation.[4] In other words, computation in $\mu\tilde{\mu}$ is the behavior that results from cutting together a compatible producer and consumer in a command, so that they may meaningfully interact with one another. In the bare $\mu\tilde{\mu}$-calculus with no logical connectives, we can only have three forms of commands: a cut between (co-)variables $\langle x \| \alpha \rangle$, a cut with an output abstraction $\langle \mu\alpha.c \| e \rangle$, and a cut with an input abstraction $\langle v \| \tilde{\mu}x.c \rangle$. In the first case, a command $\langle x \| \alpha \rangle$ represents a basic *final* state that can reduce no further, and even though its typing derivation

[4] Note, however, that the steps performed in $\mu\tilde{\mu}$ transform more of the program at once which differs from the fine-grained steps of the original cut-elimination procedures used for LK.

contains a *Cut*, it is a trivial sort of cut that corresponds more closely to a passive version of LK's *Ax*:

$$\dfrac{\dfrac{}{x : A \vdash x : A \mid}\,VR \quad \dfrac{}{\mid \alpha : A \vdash \alpha : A}\,VL}{\langle x \| \alpha \rangle : (x : A \vdash \alpha : A)}\,\mathit{Cut}$$

In the second two cases, the operational meaning of input and output abstractions are expressed via capture-avoiding substitution—much like the β law for functions in the λ-calculus—as illustrated by the following μ and $\tilde{\mu}$ rewriting rules:

$$(\mu) \qquad \langle \mu\alpha.c \| e \rangle \succ_{\mu} c\,\{e/\alpha\} \qquad\qquad (\tilde{\mu}) \qquad \langle v \| \tilde{\mu}x.c \rangle \succ_{\tilde{\mu}} c\,\{v/x\}$$

The $\tilde{\mu}$ reduction step substitutes the term v for the variable x introduced by an input abstraction, distributing it into the command c to the points where it is referenced. The μ reduction step is the mirror image, which substitutes a co-term e for a co-variable α introduced by an output abstraction. There is an extensional nature to input and output abstractions—analogous to the η law for functions in the λ-calculus—that observes the fact that trivial input and output abstractions can be eliminated by the following η_μ and $\eta_{\tilde{\mu}}$ rewriting rules:

$$(\eta_\mu) \quad \mu\alpha.\, \langle v \| \alpha \rangle \succ_{\eta_\mu} v \quad (\alpha \notin FV(v)) \quad (\eta_{\tilde{\mu}}) \quad \tilde{\mu}x.\, \langle x \| e \rangle \succ_{\eta_{\tilde{\mu}}} e \quad (x \notin FV(e))$$

In other words, the term that sends the output of v to α only to forward that information along as its own output is the same as v itself. Dually, the co-term that binds its input to x only to forward that information along to another co-term e can be written more simply as just e.

As per Remark 2.3, we can derive a reduction theory ($\twoheadrightarrow_{\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}}$) and equational theory ($=_{\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}}$) for the $\mu\tilde{\mu}$-calculus as the compatible-reflexive-transitive and compatible-reflexive-symmetric-transitive closures (respectively) of the μ, $\tilde{\mu}$, η_μ, and $\eta_{\tilde{\mu}}$ rewriting rules. It is also very easy to give the $\mu\tilde{\mu}$-calculus an operational semantics by just applying the μ and μ rewriting rules directly to commands, so that the only evaluation context is the empty context in contrast to the λ-calculus which requires deeply nested evaluation contexts. In other words a single operational reduction is given by

$$\frac{c \succ_{\mu\tilde{\mu}} c'}{c \mapsto_{\mu\tilde{\mu}} c'}$$

and multiple steps of the $\mu\tilde{\mu}$ operational semantics is the reflexive-transitive closure of the single-step $\mapsto_{\mu\tilde{\mu}}$. Note how only the operational semantics only includes the μ and $\tilde{\mu}$ rewriting rules, meaning that they are *operational rules* (Herbelin & Zimmermann, 2009). In contrast, the η_μ and $\eta_{\tilde{\mu}}$ rewriting rules are not used to run a program in the operational semantics, so they are (merely) *observational rules* meaning that the (co-)terms before and after η_μ and $\eta_{\tilde{\mu}}$ reduction are both observably the same in any program. These observational rules are never needed to run a program and get a result, because they are simulated by the operational rules whenever they come to the forefront. For example, we have the following (general) η_μ reduction which gets us to a final command:

$$\langle \mu\beta.\, \langle x \| \beta \rangle \| \alpha \rangle \to_{\eta_\mu} \langle x \| \alpha \rangle$$

But notice that in this case a μ operational step gets us to exactly the same final command anyway.

The fundamental dilemma of computation

Unfortunately, the aforementioned dynamic semantics for $\mu\tilde{\mu}$ is overly simplistic and extremely *non-deterministic*, to the point where programs may make completely divergent and unrelated computations. The non-determinism of the $\mu\tilde{\mu}$-calculus corresponds to the fact that classical cut elimination in the LK sequent calculus is also non-deterministic. The phenomenon is embodied by the fundamental conflict between input and output abstractions, as shown by the following critical pair between two dual μ and $\tilde{\mu}$ reductions for performing substitution:

$$c_1 \left\{ (\tilde{\mu}x.c_2)/\alpha \right\} \prec_\mu \langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle \succ_{\tilde{\mu}} c_2 \left\{ (\mu\alpha.c_1)/x \right\}$$

Both the term $\mu\alpha.c_1$ and co-term $\tilde{\mu}x.c_2$ are fighting for control in the above command, and either one may win. The non-deterministic outcome of this conflict is exemplified in the case where neither α nor x are referenced in their respective commands:

$$c_1 \prec_{\mu} \langle \mu\_.c_1 \| \tilde{\mu}\_.c_2 \rangle \succ_{\tilde{\mu}} c_2$$

showing that programs may produce different results each time they are run, since the same starting point may step to two different and completely arbitrary commands. This form of divergent reduction paths is called a *critical pair* and has a serious impact on the dynamic semantics of the $\mu\tilde{\mu}$-calculus.

For the $\mu\tilde{\mu}$ operational semantics, the result of a program is *non-deterministic* because it can end up in different final states depending on which rule is chosen; for example $\langle x \| \alpha \rangle \mathrel{\twoheadleftarrow\!\!\dashv}_{\mu\tilde{\mu}} \langle \mu\gamma.\, \langle x \| \alpha \rangle \| \tilde{\mu}z.\, \langle y \| \beta \rangle \rangle \mathrel{\mapstochar\twoheadrightarrow}_{\mu\tilde{\mu}} \langle y \| \beta \rangle$. This fact implies that the $\mu\tilde{\mu}\eta_{\mu}\eta_{\tilde{\mu}}$ reduction theory is not *non-confluent*, because different reductions can be applied such that the two diverging paths never converge back to the same result again. And finally, the $\mu\tilde{\mu}\eta_{\mu}\eta_{\tilde{\mu}}$ equational theory is *incoherent* because all commands and (co-)terms are equated. From the perspective of programming language semantics, this type of non-determinism can be undesirable since it makes it impossible to predict a single definitive result of a program since there may be multiple incompatible results depending on the choices made during execution. If we want to regain properties like determinism, confluence, or coherence, which are enjoyed by the λ-calculus, then some of these freedoms must be curtailed.

In order to recover determinism for the sequent calculus, Curien & Herbelin (2000) observed that we only need to choose an *evaluation strategy* that deterministically picks the next step to take by giving priority to one reduction over the other:

> Call-by-value consists in giving priority to the μ redexes, while call-by-name gives priority to the $\tilde{\mu}$ redexes.

Prioritization between the two opposed means that there must be some potential μ or $\tilde{\mu}$ redexes that we could reduce but choose not to, thereby yielding priority to the other side of the command. From another viewpoint, choosing a priority between the two sides of a command is the same thing as choosing a restriction on the terms and co-terms that can be substituted by the μ and $\tilde{\mu}$ rules. And reversing directions, choosing which terms and co-terms are substitutable by μ and $\tilde{\mu}$ reductions also chooses the evaluation strategy.

Reflecting the above observation back to the calculus, we can restore determinacy to the operational semantics and confluence to the rewriting theory by making the

$$V \in \mathit{Value}_{\mathcal{V}} ::= x \qquad\qquad E \in \mathit{CoValue}_{\mathcal{V}} ::= e$$

$$\begin{array}{llllll}
(\mu_{\mathcal{V}}) & \langle \mu\alpha.c \| E\rangle \succ_{\mu_{\mathcal{V}}} c\{E/\alpha\} & (\eta_\mu) & \mu\alpha.\,\langle v\|\alpha\rangle \succ_{\eta_\mu} v & (\alpha \notin FV(v)) \\
(\tilde{\mu}_{\mathcal{V}}) & \langle V \| \tilde{\mu}x.c\rangle \succ_{\tilde{\mu}_{\mathcal{V}}} c\{V/x\} & (\eta_{\tilde{\mu}}) & \tilde{\mu}x.\,\langle x\|e\rangle \succ_{\eta_{\tilde{\mu}}} e & (x \notin FV(e))
\end{array}$$

FIGURE 3.8. The call-by-value ($\mathcal{V}$) rewriting rules for the core $\mu\tilde{\mu}_{\mathcal{V}}$-calculus.

$$V \in \mathit{Value}_{\mathcal{N}} ::= v \qquad\qquad E \in \mathit{CoValue}_{\mathcal{N}} ::= \alpha$$

$$\begin{array}{llllll}
(\mu_{\mathcal{N}}) & \langle \mu\alpha.c \| E\rangle \succ_{\mu_{\mathcal{N}}} c\{E/\alpha\} & (\eta_\mu) & \mu\alpha.\,\langle v\|\alpha\rangle \succ_{\eta_\mu} v & (\alpha \notin FV(v)) \\
(\tilde{\mu}_{\mathcal{N}}) & \langle V \| \tilde{\mu}x.c\rangle \succ_{\mu_{\mathcal{N}}} c\{V/x\} & (\eta_{\tilde{\mu}}) & \tilde{\mu}x.\,\langle x\|e\rangle \succ_{\eta_{\tilde{\mu}}} e & (x \notin FV(e))
\end{array}$$

FIGURE 3.9. The call-by-name ($\mathcal{N}$) rewriting rules for the core $\mu\tilde{\mu}_{\mathcal{N}}$-calculus.

substitution rules strategy-aware: $\tilde{\mu}$ only substitutes *values* for variables and μ only substitutes *co-values* for co-variables. In other words, the decision of which values and co-values are substitutable is enough information to determine an evaluation strategy in the $\mu\tilde{\mu}$-calculus. To get call-by-value reduction, we can restrict the notion of value to exclude output abstractions and leave co-values unrestricted, thereby giving priority to the μ redexes as shown in Figure 3.8. Dually for call-by-name reduction, we can restrict the notion of co-value to exclude input abstractions and leave values unrestricted, thereby giving priority to the $\tilde{\mu}$ redexes as shown in Figure 3.9. Notice that in any case, the observational η_μ and $\eta_{\tilde{\mu}}$ reductions are not affected by the restrictions on (co-)values, because they do no substitution and are sound under any choice of evaluation strategy. These restrictions on substitution give us exactly Curien & Herbelin's (2000) notions of the call-by-value and call-by-name, which restores determinacy, confluence, and coherence to the dynamic semantics of $\mu\tilde{\mu}$. Excluding a (co-)term from the collection of (co-)values effectively prioritizes it by blocking opposing reductions, whereas including a (co-)term as a (co-)value diminishes its priority since it can be deleted or duplicated by substitution.

Structural rules and static scope

So far we have skirted around the issue of how the structural properties of the sequent calculus are represented in the $\mu\tilde{\mu}$-calculus. After all, they are an important part of Gentzen's LK sequent calculus, but the type system in Figure 3.7 does not express them. For instance, the co-term $\tilde{\mu}z.\,\langle x\|\alpha\rangle$ should have the type $x : X \mid \tilde{\mu}z.\,\langle x\|\alpha\rangle : Y \vdash \alpha : X$, but there's no way to derive that conclusion with the typing rules in Figure 3.7 alone. What's missing here is a way to infer weakening on the left, which is a symptom of the general lack of structural properties in the raw core typing rules. There are multiple options for restoring the classical structural properties to the core $\mu\tilde{\mu}$ type system, and to be thorough we will compare two of the most commonly used methods. The common theme behind both methods is to equate the structural properties of sequents with the scoping properties of static variables and co-variables in expressions.

The first method of expressing the structural properties of sequents in $\mu\tilde{\mu}$ is to add explicit structural rules that allow for a single (co-)variable to appear any number of times in an expression. The full collection of these structural scoping rules are shown in Figure 3.10, which corresponds one-for-one with the structural rules of Gentzen's LK sequent calculus over each form of $\mu\tilde{\mu}$ expression. The weakening rules say that even if a free (co-)variable is in scope in an expression, it does not have to be referenced, as in the co-term $\tilde{\mu}z.\,\langle x\|\alpha\rangle$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : X \vdash x : X \mid}\,\mathit{VR} \qquad \dfrac{}{\mid \alpha : X \vdash \alpha : X}\,\mathit{VL}}{\langle x\|\alpha\rangle : (x : X \vdash \alpha : X)}\,\mathit{Cut}}{\langle x\|\alpha\rangle : (x : X, z : Y \vdash \alpha : X)}\,\mathit{WL}}{x : X \mid \tilde{\mu}z.\,\langle x\|\alpha\rangle : Y \vdash \alpha : X}\,\mathit{AL}$$

The contraction rules say that a free (co-)variable can be referenced an additional time by replacing another (co-)variable, as in the command $\langle \mu\delta.\,\langle y\|\alpha\rangle\|\tilde{\mu}z.\,\langle y\|\alpha\rangle\rangle$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : X \vdash y : X \mid}\,\mathit{VR} \qquad \dfrac{}{\mid \beta : X \vdash \beta : X}\,\mathit{VL}}{\langle y\|\beta\rangle : (y : X \vdash \beta : X)}\,\mathit{Cut}}{\langle y\|\beta\rangle : (y : X \vdash \delta : Y, \beta : X)}\,\mathit{WR}}{y : X \vdash \mu\delta.\,\langle y\|\beta\rangle : Y \mid \beta : X}\,\mathit{AR} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x : X \vdash x : X \mid}\,\mathit{VR} \qquad \dfrac{}{\mid \alpha : X \vdash \alpha : X}\,\mathit{VL}}{\langle x\|\alpha\rangle : (x : X \vdash \alpha : X)}\,\mathit{Cut}}{\langle x\|\alpha\rangle : (x : X, z : Y \vdash \alpha : X)}\,\mathit{WL}}{x : X \mid \tilde{\mu}z.\,\langle x\|\alpha\rangle : Y \vdash \alpha : X}\,\mathit{AL}}{\langle \mu\delta.\,\langle y\|\beta\rangle\|\tilde{\mu}z.\,\langle x\|\alpha\rangle\rangle : (x : X, y : X \vdash \alpha : X, \beta : X)}\,\mathit{Cut}}{\langle \mu\delta.\,\langle x\|\beta\rangle\|\tilde{\mu}z.\,\langle x\|\alpha\rangle\rangle : (x : X \vdash \alpha : X, \beta : X)}\,\mathit{CL}$$

$$\frac{c : (\Gamma \vdash \Delta)}{c : (\Gamma \vdash \alpha : A, \Delta)}\ \mathit{WR} \qquad \frac{c : (\Gamma \vdash \Delta)}{c : (\Gamma, x : A \vdash \Delta)}\ \mathit{WL}$$

$$\frac{c : (\Gamma \vdash \alpha : A, \beta : A, \Delta)}{c\,\{\alpha/\beta\} : (\Gamma \vdash \alpha : A, \Delta)}\ \mathit{CR} \qquad \frac{c : (\Gamma, y : A, x : A \vdash \Delta)}{c\,\{x/y\} : (\Gamma, x : A \vdash \Delta)}\ \mathit{CL}$$

$$\frac{c : (\Gamma \vdash \Delta, \alpha : A, \beta : B, \Delta')}{c : (\Gamma \vdash \Delta, \beta : B, \alpha : A, \Delta')}\ \mathit{XR} \qquad \frac{c : (\Gamma', y : B, x : A, \Gamma \vdash \Delta)}{c : (\Gamma', x : A, y : B, \Gamma \vdash \Delta)}\ \mathit{XL}$$

$$\frac{\Gamma \vdash v : C \mid \Delta}{\Gamma \vdash v : C \mid \alpha : A, \Delta}\ \mathit{WR} \qquad \frac{\Gamma \vdash v : C \mid \Delta}{\Gamma, x : A \vdash v : C \mid \Delta}\ \mathit{WL}$$

$$\frac{\Gamma \vdash v : C \mid \alpha : A, \beta : A, \Delta}{\Gamma \vdash v\,\{\alpha/\beta\} : C \mid \alpha : A, \Delta}\ \mathit{CR} \qquad \frac{\Gamma, y : A, x : A \vdash v : C \mid \Delta}{\Gamma, x : A \vdash v\,\{x/y\} : C \mid \Delta}\ \mathit{CL}$$

$$\frac{\Gamma \vdash v : C \mid \Delta, \alpha : A, \beta : B, \Delta'}{\Gamma \vdash v : C \mid \Delta, \beta : B, \alpha : A, \Delta'}\ \mathit{XR} \qquad \frac{\Gamma', y : B, x : A, \Gamma \vdash v : C \mid \Delta}{\Gamma', x : A, y : B, \Gamma \vdash v : C \mid \Delta}\ \mathit{XL}$$

$$\frac{\Gamma \mid e : C \vdash \Delta}{\Gamma \mid e : C \vdash \alpha : A, \Delta}\ \mathit{WR} \qquad \frac{\Gamma \mid e : C \vdash \Delta}{\Gamma, x : A \mid e : C \vdash \Delta}\ \mathit{WL}$$

$$\frac{\Gamma \mid e : C \vdash \alpha : A, \beta : A, \Delta}{\Gamma \mid e\,\{\alpha/\beta\} : C \vdash \alpha : A, \Delta}\ \mathit{CR} \qquad \frac{\Gamma, y : A, x : A \mid e : C \vdash \Delta}{\Gamma, x : A \mid e\,\{x/y\} : C \vdash \Delta}\ \mathit{CL}$$

$$\frac{\Gamma \mid e : C \vdash \Delta, \alpha : A, \beta : B, \Delta'}{\Gamma \mid e : C \vdash \Delta, \beta : B, \alpha : A, \Delta'}\ \mathit{XR} \qquad \frac{\Gamma', y : B, x : A, \Gamma \mid e : C \vdash \Delta}{\Gamma', x : A, y : B, \Gamma \mid e : C \vdash \Delta}\ \mathit{XL}$$

FIGURE 3.10. Scoping rules for (co-)variables in commands, terms, and co-terms.

Finally, the exchange rules say that the order of the (co-)variables in scope does not matter. Notice that none of these rules are syntactically visible in their expression. Unlike the axiom, activation, and cut rules that only apply to expressions starting with a very specific form, the structural rules could potentially apply to expressions of any form so they are not directed by syntax.

The scoping rules in Figure 3.10 can seem repetitive or even redundant: the same weakening, contraction, and exchange rules are repeated three times for commands, terms, and co-terms. Indeed, with this style of presenting the structural properties of sequents, it is common to limit the rules to a single form of expression like commands (Wadler, 2003; Munch-Maccagnoni, 2009). Unfortunately however, the repetition for each kind of expression and sequent is necessary to ensure that the structural rules match our expectation of static scope in programming languages. For example, in anticipation of the imminent extension of $\mu\tilde{\mu}$ with function types in Section 3.3, we might want to call a binary function of type $X \to X \to Y$ with the same value for both arguments, as in the co-term $x \cdot x \cdot \beta$. To type this co-term, we need to contract x in the co-term itself, as in:

$$\dfrac{\dfrac{\dfrac{}{x' : X \vdash x' : X \mid}\,VR \qquad \dfrac{\dfrac{}{x : X \vdash x : X \mid}\,VR \quad \dfrac{}{\mid \beta : Y \vdash \beta : Y}\,VL}{x : X \mid x \cdot \beta : X \to Y \vdash \beta : Y}\,{\to}L}{x' : X, x : X \mid x' \cdot x \cdot \beta : X \to X \to Y \vdash \beta : Y}\,{\to}L}{x : X \mid x \cdot x \cdot \beta : X \to X \to Y \vdash \beta : Y}\,CL$$

which is not possible if we only allow contraction in commands. Furthermore, only including the structural rules for commands can mean that sensible observational reductions like η_μ and $\eta_{\tilde{\mu}}$ no longer preserve the type of expressions. For example, the η_μ-expanded term $\mu\alpha. \langle x \| \alpha \rangle$ can be assigned the type $y : Y, x : X \vdash \mu\alpha. \langle x \| \alpha \rangle : X \mid \beta : Y$ using weakening and exchange on commands as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : X \vdash x : X \mid}\,VR \quad \dfrac{}{\mid \alpha : X \vdash \alpha : X}\,VL}{\langle x \| \alpha \rangle : (x : X \vdash \alpha : X)}\,Cut}{\langle x \| \alpha \rangle : (x : X \vdash \beta : Y, \alpha : X)}\,WR}{\langle x \| \alpha \rangle : (x : X \vdash \alpha : X, \beta : Y)}\,XR}{\langle x \| \alpha \rangle : (x : X, y : Y \vdash \alpha : X, \beta : Y)}\,WL}{\langle x \| \alpha \rangle : (y : Y, x : X \vdash \alpha : X, \beta : Y)}\,XL}{y : Y, x : X \vdash \mu\alpha. \langle x \| \alpha \rangle : X \mid \beta : Y}\,AR$$

But there is no way to conclude $y : Y, x : X \vdash x : X \mid \beta : Y$ without the structural rules for terms, even though it is a reduct of a term of that type: $\mu\alpha. \langle x \| \alpha \rangle \to_{\eta_\mu} x$.

The second method of expressing the structural properties of sequents in $\mu\tilde{\mu}$ is by treating the environments Γ and Δ as unordered sets associating types to (co-)variables and generalizing the axiom and cut rules to implicitly accomodate several steps of weakening and contraction, respectively (Curien & Herbelin, 2000; Wadler, 2005; Munch-Maccagnoni, 2013). This extension of the core $\mu\tilde{\mu}$ type system is shown in Figure 3.11 and corresponds to the variant of LK with implicit structural rules discussed in Remark 3.2. In this formulation, there is no explicit use of structural rules in a typing derivation, but instead the structural properties of sequents follow

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ VR \qquad \frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}\ VL$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ AR \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\ AL$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle v \| e \rangle : (\Gamma \vdash \Delta)}\ Cut$$

FIGURE 3.11. Implicit (co-)variable scope in the core $\mu\tilde{\mu}$ typing.

from the natural scoping rules for static (co-)variables in the $\mu\tilde{\mu}$-calculus, analogous to the scoping rules for the λ-calculus. During type checking, an output abstraction $\Gamma \vdash \mu\alpha.c : A \mid \Delta$ (and dually an input abstraction $\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta$) signals that the active type A may undergo an arbitrary number of structural rules depending on how α (dually x) is referenced in c. During execution, the behavior of structural rules are implicitly implemented by the substitution operation used by μ and $\tilde{\mu}$ reduction, corresponding to the structural steps of a cut elimination procedure.

As stated before for the logic of LK in Remark 3.2, the choice between the two formulations of the scoping properties of $\mu\tilde{\mu}$ (co-)variables is somewhat arbitrary and a matter of taste. Since we are dealing with a calculus corresponding to classical logic, both treatments of structural properties are equivalent to each other in a sense—both formulations will admit type checking the same expressions, even in richer extensions of the core language. However, the two formulations have their own advantages. The implicit scoping presented in Figure 3.11 is concise and forgoes the redundancy of repeated rules, whereas the explicit scoping presented in Figure 3.10 easily allows for a more refined analysis of the structural properties and exploration of sub-structural calculi (Munch-Maccagnoni, 2009) corresponding to sub-structural logics that forbid certain uses of structural rules. The most important thing, though, is that *something* is done to express the scope of (co-)variables in the classical language $\mu\tilde{\mu}$. For our purposes here, we will take the explicit formulation of scoping rules in Figure 3.10 as the canonical definition for classical $\mu\tilde{\mu}$ in the remainder.

Remark 3.6. As it turns out, output abstractions in the $\mu\tilde{\mu}$-calculus let programs manipulate their own control flow similar to Scheme's (Kelsey *et al.*, 1998) callcc control operator, or Felleisen's (1992) $\mathcal{C}$ operator. Intuitively, a use of callcc or an abort can be read in terms of an output abstraction that duplicates or deletes its bound co-variable, respectively:

$$\mathsf{callcc}(\lambda\alpha.v) \triangleq \mu\alpha.\, \langle v \| \alpha\rangle \qquad \mathsf{abort}\, c \triangleq \mu\delta.c \qquad (\delta \notin FV(c))$$

This phenomenon is a consequence of Griffin's (1990) observation that under the Curry-Howard correspondence, classical logic corresponds to control flow manipulation, along with the fact that the LK sequent calculus formalizes classical logic (see Remark 3.5). Under this interpretation, multiple consequences in the sequent calculus correspond to multiple available co-variables which give the program multiple possible exit paths. The weakening and contraction rules on the right for these multiple consequences correspond to deleting or copying an exit path, respectively. Indeed, multiple consequences with right-handed structural rules may be seen as the logical essence for this "classical" form of control effects (so called for the connection to classical logic as well as callcc being the traditional control operator), since extending natural deduction with multiple consequences, as in Parigot's (1992) $\lambda\mu$-calculus, gives rise to a programming language with control effects equivalent to callcc (Ariola & Herbelin, 2003). *End remark* 3.6.

The Dual Calculi

With the core $\mu\tilde{\mu}$ language firmly in place, we can now enrich it with additional programming constructs that correspond to the logical elements—the connectives and logical rules—of Gentzen's LK sequent calculus. The syntax and typing rules for these extra logical constructs are shown in Figure 3.12,[5] which extends the core $\mu\tilde{\mu}$-calculus from Figure 3.7 along with the structural (co-)variable scoping rules from Figure 3.10. This language combines both Curien & Herbelin's (2000) $\overline{\lambda}\mu\tilde{\mu}$-calculus (the portion associated with implication) and Wadler's (2003) dual calculus (the portion associated with conjunction, disjunction, and negation) into a single calculus corresponding to all of the simply-typed LK sequent calculus. Furthermore, the quantifiers of LK are interpreted as a sequent calculus version of system F (Reynolds, 1983; Girard *et al.*, 1989): universal quantification ($\forall$) acts as an abstraction over types analogous to implication, and existential quantification ($\exists$) is the mirror image of $\forall$. We refer to

[5] To help syntactically distinguish terms from co-terms, we use the notational convention throughout that round parentheses are the grouping brackets for terms, and square brackets are the grouping brackets for co-terms.

this combined language here as the “dual calculi” because, as we will soon see, the language is the basis for two different but highly related calculi that exhibit dual computational behavior to one another.

Since the right introduction rules for logical connectives are shared by both natural deduction and the sequent calculus, the dual calculi terms for creating results of product, sum, and function types have the same form as in the λ-calculus. Products are introduced by pairing, (v, v'), sums are introduced by injection, $\iota_1(v)$ and $\iota_2(v)$, and functions are introduced by λ-abstractions, $\lambda x.v$. Additionally, the terms for creating results of universally quantified types are Λ-abstractions, $\Lambda X.v$, as in system F, and the results of existentially quantified types are “masked” terms, $B \mathbin{@} v$, that hide the type B in the underlying term v from being visible from the outside. In contrast, the left introduction rules of the sequent calculus are distinct from the right elimination rules of natural deduction, so the difference between the λ-calculus and the dual calculi really appears when results are used.

Instead of function application, the left implication introduction $\to L$ builds a co-term that represents a *call-stack*. If v is a term that produces a result of type A, and e is a co-term that consumes a result of type B, then the call-stack $v \cdot e$ is a co-term that works with a function value of type $A \to B$ by feeding it v as an argument and sending the returned result to e. For example, given that $x_1 : A_1$, $x_2 : A_2$, $x_3 : A_3$, and $\beta : B$, then the call-stack $x_1 \cdot [x_2 \cdot [x_3 \cdot \beta]]$ is expecting to consume a function of type $A_1 \to (A_2 \to (A_3 \to B))$:[6]

$$\dfrac{\dfrac{}{x_1{:}A_1 \vdash x_1 : A_1 \mid}\mathit{VR} \quad \dfrac{\dfrac{}{x_2{:}A_2 \vdash x_2 : A_2 \mid}\mathit{VR} \quad \dfrac{\dfrac{}{x_3{:}A_3 \vdash x_3 : A_3 \mid}\mathit{VR} \quad \dfrac{}{\mid \beta : B \vdash \beta{:}B}\mathit{VL}}{x_3{:}A_3 \mid x_3 \cdot \beta : A_3 \to B \vdash \beta : B}{\to}L}{x_3{:}A_3, x_2{:}A_2 \mid x_2 \cdot x_3 \cdot \beta : A_2 \to A_3 \to B \vdash \beta{:}B}{\to}L}{x_3{:}A_3, x_2{:}A_2, x_1{:}A_1 \mid x_1 \cdot x_2 \cdot x_3 \cdot \beta : A_1 \to A_2 \to A_3 \to B \vdash \beta{:}B}{\to}L$$

The left introductions for the other type constructors follow a similar pattern, with each one building a co-term that expects to consume a value of that type. There are two left conjunction introductions corresponding to the two projections out of a product. If e_1 is a co-term that consumes a value of type A, then $\times L_1$ builds the

[6] Like the common notational convention in the simply-typed λ-calculus that the function type constructor associates to the right, so that $A_1 \to A_2 \to A_3 \to B = A_1 \to (A_2 \to (A_3 \to B))$, we adopt a similar notational convention that the call stack constructor associates to the right, so that $x_1 \cdot x_2 \cdot x_3 \cdot \beta = x_1 \cdot [x_2 \cdot [x_3 \cdot \beta]]$.

$$\begin{aligned}
A, B, C \in \mathit{Type} &::= X \mid A \times B \mid A + B \mid \neg A \mid A \to B \mid \forall X.A \mid \exists X.A \\
c \in \mathit{Command} &::= \langle v \| e \rangle \\
v \in \mathit{Term} &::= x \mid \mu\alpha.c \mid (v, v) \mid \iota_1\,(v) \mid \iota_2\,(v) \mid \mathsf{not}(e) \mid \lambda x.v \mid \Lambda X.v \mid B \mathbin{@} v \\
e \in \mathit{CoTerm} &::= \alpha \mid \tilde{\mu}x.c \mid \pi_1\,[e] \mid \pi_2\,[e] \mid [e, e] \mid \mathsf{not}[v] \mid v \cdot e \mid B \mathbin{@} e \mid \tilde{\Lambda}X.e \\
\Gamma \in \mathit{InputEnv} &::= x_1 : A_1, \ldots, x_n : A_n \\
\Delta \in \mathit{OutputEnv} &::= \alpha_1 : A_2, \ldots, \alpha_n : A_n \\
\mathit{Judgement} &::= c : (\Gamma \vdash \Delta) \mid (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta)
\end{aligned}$$

Logical rules:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash v' : B \mid \Delta}{\Gamma \vdash (v, v') : A \times B \mid \Delta} \times R$$

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \pi_1\,[e] : A \times B \vdash \Delta} \times L_1 \qquad \frac{\Gamma \mid e : B \vdash \Delta}{\Gamma \mid \pi_2\,[e] : A \times B \vdash \Delta} \times L_2$$

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_1\,(v) : A + B \mid \Delta} +R_1 \qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \iota_2\,(v) : A + B \mid \Delta} +R_2$$

$$\frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid e' : B \vdash \Delta}{\Gamma \mid [e, e'] : A + B \vdash \Delta} +L$$

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash \mathsf{not}(e) : \neg A \mid \Delta} \neg R \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \mid \mathsf{not}[v] : \neg A \vdash \Delta} \neg L$$

$$\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta} \to R \qquad \frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : B \vdash \Delta'}{\Gamma', \Gamma \mid v \cdot e : A \to B \vdash \Delta', \Delta} \to L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \Lambda X.v : \forall X.A \mid \Delta} \forall R \qquad \frac{\Gamma \mid e : A\,\{B/X\} \vdash \Delta}{\Gamma \mid B \mathbin{@} e : \forall X.A \vdash \Delta} \forall L$$

$$\frac{\Gamma \vdash v : A\,\{B/X\} \mid \Delta}{\Gamma \vdash B \mathbin{@} v : \exists X.A \mid \Delta} \exists R \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\Lambda}X.e : \exists X.A \vdash \Delta} \exists L$$

FIGURE 3.12. The syntax and types for the dual calculi.

co-term $\pi_1 [e_1]$ that works with a value of type $A \times B$ by projecting out the first element of the product and sending it to e_1 when needed (and similarly for the second projection $\pi_2 [e_2]$ built by $\times L_2$). If e_1 and e_2 are co-terms that consume values of type A and B, respectively, then $+L$ builds the co-term $[e_1, e_2]$ that works with a value of type $A + B$ by checking its constructor: an injection of the form $\iota_1 (v_1)$ has the value of v_1 sent to e_1 as needed, and likewise an injection of the form $\iota_2 (v_2)$ has the value of v_2 sent to e_2 as needed. The co-term for $\forall L$ is similar to the call stacks of $\to L$, so that if e is a co-term that consumes a value at the particular type $A \{B/X\}$, then $B @ e$ works with a value of the general type $\forall X.A$ by first specializing the polymorphic value and then passing it along to e. Perhaps the most unusual co-term comes from $\exists L$, but this is just the mirror image of the $\forall R$ term. If e is a co-term that consumes a value of type A, containing a generic type variable X, then $\exists L$ gives the abstracted co-term $\tilde{\Lambda} X.e$ that works with a value of type $\exists X.A$ by instantiating X with the value's hidden type before passing the underlying value to e.

The one type constructor that is not typically found in the λ-calculus, but commonly in a sequent calculus like LK or the dual calculi, is negation. The negation type $\neg A$ represents an inversion between producers and consumers—terms and co-terms—during computation. Intuitively, negation expresses a form of *continuations*: a term of type $\neg A$ is actually a consumer of A. The right negation introduction allows terms to contain consumers, so that if e is a co-term expecting an input a result of type A then $\neg R$ builds the term $\mathsf{not}(e)$. Dually, the left negation introduction allows co-terms to contain producers, so that if v is a term expecting to output a result of type A then $\neg L$ builds the co-term $\mathsf{not}[v]$. When a negated term and co-term meet each other in a command, the inversion is undone so that their underlying components change places and continue the interaction.

The above intuition on the dynamic meaning of types in the dual calculi can be codified into rewriting rules. Recall from Section 3.2 that the semantics of the core $\mu\tilde{\mu}$-calculus was split in two to restore determinacy and confluence: one corresponding to call-by-value and the other to call-by-name. Likewise, there are two semantics for the dual calculi, so that the same language bears two different calculi (hence the name). Since both semantics of the core $\mu\tilde{\mu}$-calculus are already given in Figure 3.8 and Figure 3.9, we only need to suitably expand the notions of value and co-value to accomodate the new (co-)term introductions and explain the logical steps of cut elimination (referred to by the common name β) that occur when two opposed

$$\begin{aligned} V \in \mathit{Value}_{\mathcal{V}} &::= x \mid (V, V) \mid \iota_1(V) \mid \iota_2(V) \mid \mathsf{not}(e) \mid \lambda x.v \mid \Lambda X.v \mid A \mathbin{@} V \\ E \in \mathit{CoValue}_{\mathcal{V}} &::= e \end{aligned}$$

$$\begin{array}{llll}
(\beta^{\times}_{\mathcal{V}}) & \langle (V_1, V_2) \| \pi_i [E] \rangle \succ_{\beta^{\times}_{\mathcal{V}}} \langle V_i \| E \rangle & (\beta^{+}_{\mathcal{V}}) & \langle \iota_i(V) \| [E_1, E_2] \rangle \succ_{\beta^{+}_{\mathcal{V}}} \langle V \| E_i \rangle \\
(\beta^{\neg}_{\mathcal{V}}) & \langle \mathsf{not}(e) \| \mathsf{not}[v] \rangle \succ_{\beta^{\neg}_{\mathcal{V}}} \langle v \| e \rangle & (\beta^{\to}_{\mathcal{V}}) & \langle \lambda x.v \| V \cdot E \rangle \succ_{\beta^{\to}_{\mathcal{V}}} \langle v\{V/x\} \| E \rangle \\
(\beta^{\forall}_{\mathcal{V}}) & \langle \Lambda X.v \| B \mathbin{@} E \rangle \succ_{\beta^{\forall}_{\mathcal{V}}} \langle v\{B/X\} \| E \rangle & (\beta^{\exists}_{\mathcal{V}}) & \left\langle B \mathbin{@} V \middle\| \tilde{\Lambda} X.e \right\rangle \succ_{\beta^{\exists}_{\mathcal{V}}} \langle V \| e\{B/X\} \rangle
\end{array}$$

FIGURE 3.13. The β laws for the call-by-value ($\mathcal{V}$) half of the dual calculi.

$$\begin{aligned} V \in \mathit{Value}_{\mathcal{N}} &::= v \\ E \in \mathit{CoValue}_{\mathcal{N}} &::= \alpha \mid \pi_1[E] \mid \pi_2[E] \mid [E, E] \mid \mathsf{not}(v) \mid v \cdot E \mid B \mathbin{@} E \mid \tilde{\Lambda} X.e \end{aligned}$$

$$\begin{array}{llll}
(\beta^{\times}_{\mathcal{V}}) & \langle (V_1, V_2) \| \pi_i [E] \rangle \succ_{\beta^{\times}_{\mathcal{V}}} \langle V_i \| E \rangle & (\beta^{+}_{\mathcal{V}}) & \langle \iota_i(V) \| [E_1, E_2] \rangle \succ_{\beta^{+}_{\mathcal{V}}} \langle V \| E_i \rangle \\
(\beta^{\neg}_{\mathcal{V}}) & \langle \mathsf{not}(e) \| \mathsf{not}[v] \rangle \succ_{\beta^{\neg}_{\mathcal{V}}} \langle v \| e \rangle & (\beta^{\to}_{\mathcal{V}}) & \langle \lambda x.v \| V \cdot E \rangle \succ_{\beta^{\to}_{\mathcal{V}}} \langle v\{V/x\} \| E \rangle \\
(\beta^{\forall}_{\mathcal{V}}) & \langle \Lambda X.v \| B \mathbin{@} E \rangle \succ_{\beta^{\forall}_{\mathcal{V}}} \langle v\{B/X\} \| E \rangle & (\beta^{\exists}_{\mathcal{V}}) & \left\langle B \mathbin{@} V \middle\| \tilde{\Lambda} X.e \right\rangle \succ_{\beta^{\exists}_{\mathcal{V}}} \langle V \| e\{B/X\} \rangle
\end{array}$$

FIGURE 3.14. The β laws for the call-by-name ($\mathcal{N}$) half of the dual calculi.

introduction forms of the same type meet in a command. The call-by-value β rules are given in Figure 3.13 and the call-by-name β rules are given in Figure 3.14, both of which extend the core semantics from Figure 3.8 and Figure 3.9, respectively. The $\beta^{\times}$, β^{+} and $\beta^{\neg}$ rules come from Wadler's (2003) dual calculus whereas the $\beta^{\to}$ rules are inspired by Curien & Munch-Maccagnoni's (2010) revision of the $\overline{\lambda}\mu\tilde{\mu}$-calculus.

The β laws extend the previous dynamic semantics of the core $\mu\tilde{\mu}$-calculus to account for the additional programming constructs. As per Remark 2.3, we have a reduction theory ($\twoheadrightarrow_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}}$), equational theory ($=_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}}$), and an operational semantics ($\mapsto\!\!\!\!\twoheadrightarrow_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}}$) for the call-by-value dual calculus from the $\mu_{\mathcal{V}}$, $\tilde{\mu}_{\mathcal{V}}$, η_{μ}, $\eta_{\tilde{\mu}}$, and $\beta_{\mathcal{V}}$ laws, as well as a reduction theory ($\twoheadrightarrow_{\mu_{\mathcal{N}}\mu_{\mathcal{N}}\beta_{\mathcal{N}}}$), equational theory ($=_{\mu_{\mathcal{N}}\mu_{\mathcal{N}}\beta_{\mathcal{N}}}$), and operational semantics ($\mapsto\!\!\!\!\twoheadrightarrow_{\mu_{\mathcal{N}}\mu_{\mathcal{N}}\beta_{\mathcal{N}}}$) for the call-by-name dual calculus from the $\mu_{\mathcal{N}}$, $\tilde{\mu}_{\mathcal{N}}$, η_{μ}, $\eta_{\tilde{\mu}}$, and $\beta_{\mathcal{N}}$ laws. As before, both the call-by-value and call-by-name operational semantics applies the rewriting rules directly to commands.

Notice that, like in the core $\mu\tilde{\mu}$-calculus, the form of the operational β rules are the same in both semantics, so that the only difference is the definition of *value* and *co-value* referred to in those rules. The rule of thumb is that a β rule only applies when an introductory value and co-value interact in a command. For example, the call-by-value $\beta_{\mathcal{V}}^{\times}$ rule will only project from a pair value to extract a component that is also a value. These restrictions are captured in the call-by-value definition of value that admits only "simple" terms and hereditarily excludes complex terms like $\mu\alpha.c$ (representing an arbitrarily complex computation before yielding a result on α) from the values of product and sum types, which matches the behavior of products and sums in strict functional languages like ML. However, there is no such restriction on co-terms in the call-by-value operational semantics, so any co-term counts as a co-value. Dually, the call-by-name $\beta_{\mathcal{N}}^{\times}$ rule will only project out of a pair when it is needed by a projection co-value to send that component the underlying co-value. These restrictions are captured in the call-by-name definition of co-value that admits only "strict" co-terms and hereditarily excludes complex co-terms like $\tilde{\mu}x.c$ (representing an arbitrarily complex computation before demanding a result for x) from the co-values of product and sum types. However, there is no restriction on terms in the call-by-name operational semantics, so any term counts as a value.

Remark 3.7. It's worthwhile to mention that although the dual calculi are primarily seen as typed languages, their semantics do not use any type information to run commands. We can therefore execute untyped commands as well as typed ones, which of course creates the possibility of getting stuck at fatal type errors. Untyped commands also open up the possibility of running general recursive programs, which can be encoded in a similar manner as in the λ-calculus without any additional features of the language. For example, Curry's untyped fixed-point Y combinator in the λ-calculus:

$$Y \triangleq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

can be analogously defined in the dual calculi using functions as:

$$Y \triangleq \lambda f.\mu\alpha.\left\langle \lambda x.\mu\beta.\left\langle f \middle\| \mu\gamma.\left\langle x \middle\| x \cdot \gamma\right\rangle \cdot \beta\right\rangle \middle\| (\lambda x.\mu\beta.\left\langle f \middle\| \mu\gamma.\left\langle x \middle\| x \cdot \gamma\right\rangle \cdot \beta\right\rangle) \cdot \alpha\right\rangle$$

The two share analogous behavior: in the λ-calculus $Y\ f = f\ (Y\ f)$ and in the dual calculi $\langle Y \| f \cdot \alpha\rangle = \langle f \| \mu\beta.\langle Y \| f \cdot \beta\rangle \cdot \alpha\rangle$. Also analogous to the non-terminating untyped term $\Omega \triangleq (\lambda x.x\ x)\ (\lambda x.x\ x)$ in the λ-calculus, the dual calculi both have non-terminating untyped commands, which can be written using functions or more simply with negation:

$$\Omega \triangleq \langle \mathsf{not}(\tilde{\mu}x.\, \langle x \| \mathsf{not}[x] \rangle) \| \mathsf{not}[\mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle] \rangle$$

For example, in the call-by-name operational semantics, we have the following infinite execution of Ω:

$$\begin{aligned}
\Omega &\triangleq \langle \mathsf{not}(\tilde{\mu}x.\, \langle x \| \mathsf{not}[x] \rangle) \| \mathsf{not}[\mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle] \rangle \\
&\mapsto_{\beta^{\neg}_{\mathcal{N}}} \langle \mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle \| \tilde{\mu}x.\, \langle x \| \mathsf{not}[x] \rangle \rangle \\
&\mapsto_{\tilde{\mu}_{\mathcal{N}}} \langle \mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle \| \mathsf{not}[\mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle] \rangle \\
&\mapsto_{\mu_{\mathcal{N}}} \langle \mathsf{not}(\mathsf{not}[\mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle]) \| \mathsf{not}[\mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle] \rangle \\
&\mapsto_{\beta^{\neg}_{\mathcal{N}}} \langle \mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle \| \mathsf{not}[\mu\alpha.\, \langle \mathsf{not}(\alpha) \| \alpha \rangle] \rangle \\
&\mapsto_{\mu_{\mathcal{N}}} \dots
\end{aligned}$$

Note that encoding general recursion in the untyped sequent calculus requires *some* logical connective, like negation or implication. The core $\mu\tilde{\mu}$-calculus gives a more restrained language of substitution that does not express general recursion even in the untyped calculus, where general (and non-confluent) μ- and $\tilde{\mu}$-reduction is still *strongly normalizing* (Polonovski, 2004)—that is, there are no infinite sequences of $\mu\tilde{\mu}$-reductions. This fact is in contrast with the untyped λ-calculus which can express general recursion, because β-reduction is not strongly normalizing in the untyped calculus. *End remark* 3.7.

Focusing on computation

There is a problem lurking in the β-based operational semantics for the dual calculi. Consider how we would evaluate the projection $\pi_1((f\ 1), 2)$ in a call-by-value functional language like ML. First we would compute the application $f\ 1$ to construct the pair value, then we would compute the π_1 projection of that pair and extract the value returned by $f\ 1$ as the result of the expression. However, if we represent this program as the following command in the call-by-value dual calculus:[7]

$$\langle ((\mu\beta.\, \langle f \| 1 \cdot \beta \rangle), 2) \| \pi_1\, [\alpha] \rangle$$

we find that no operational rule matches this command, so we are stuck! This isn't just a problem with the call-by-value operational semantics. The command:

$$\langle (1, 2) \| \pi_1\, [\tilde{\mu}x.\, \langle 0 \| \alpha \rangle] \rangle$$

which corresponds to the expression $\mathbf{let}\, x = \pi_1(1, 2)\, \mathbf{in}\, 0$ in a functional language, is also stuck in the call-by-name operational semantics.

This is clearly an undesirable situation that breaks the connection between the λ-calculus and dual calculi—we should not get stuck on such commands with unfinished computation in introduction forms—so something needs to be done to refocus the attention in a command to the next step of computation. As it stands now in the dual calculi, we either have too many programs with unexplained behavior, or too few behaviors for executing programs. Correspondingly, there are two general techniques to remedy prematurely stuck commands and restore the connection between λ-calculus and the dual calculi:

(1) The *static* approach (Curien & Herbelin, 2000) removes the superfluous parts of the syntax that cause β reduction to get stuck, but are not necessary to express all the same computations as the original language.

(2) The *dynamic* approach (Wadler, 2003) adds the necessary extra steps to the operational semantics that *lift* buried computations to the top of the command, so that they are exposed and may take over control of the computation.

Both of these techniques are an application of an idea called *focusing* (Andreoli, 1992; Laurent, 2002) from proof search at different points in a programs life—either at "run time" or at "compile time"—to make sure that the call-by-value and call-by-name semantics are complete without missing out on any essential capabilities of the language.

[7] Here, α stands for the empty, or top-level, context which is implicit in the functional expression.

Static focusing

For the static method of focusing, consider which syntactic patterns could lead to β-stuck commands. In the call-by-value command above, $\langle ((\mu\beta.\, \langle f \| 1 \cdot \beta \rangle), 2) \| \pi_1 [\alpha] \rangle$, the problem is that a pair with a non-value component (namely the first one) is interacting with a projection co-value. Because the pair does not have values for both components, the $\beta_{\mathcal{V}}^{\times}$ operational step does not apply. Dually, the call-by-name command above, $\langle (1, 2) \| \pi_1 [\tilde{\mu} x.\, \langle 0 \| \alpha \rangle] \rangle$, puts a pair value in interaction with a projection that has a non-co-value component. Because the projection does not contain a co-value, the $\beta_{\mathcal{N}}^{\times}$ operational step does not apply. After examining all the $\beta_{\mathcal{V}}$ rules, we see that the call-by-value $\beta_{\mathcal{V}}$ operational semantics is only equipped to deal with certain introduction forms containing values (namely the pairing $\times R$, injection $+R$, and masking $\exists R$ terms as well as calling $\to L$ co-terms). Similarly, the call-by-name $\beta_{\mathcal{N}}$ operational semantics is only equipped to deal with certain introduction co-terms containing co-values (namely the projection $\times L$, matching $+L$, and calling $\to L$, and specializing $\forall L$ co-terms).

We can rule out the problematic commands via static focusing by limiting ourselves to a sub-syntax of the dual calculi. However, since each operational semantics (both call-by-value and call-by-name) have difficulty with different parts of the syntax, static focusing effectively splits the language in two: one sub-syntax for each evaluation strategy. For call-by-value, we must bake in the notion of values into the syntax and restrict the $\times R$, $+R$, $\exists R$, and $\to L$ inference rules appropriately. Doing so gives us the LKQ sub-calculus (Curien & Herbelin, 2000) shown in Figure 3.15. Dually for call-by-name, we must bake in the notion of co-values into the syntax and restrict the $\times L$, $+L$, $\to L$, and $\forall L$ inference rules appropriately, giving the LKT sub-calculus shown in Figure 3.16.

The associated type systems separate the restricted notions of (co-)values from general (co-)terms through a new form of *focused* sequent with a stricter sense of active formula held in a *stoup* (Girard, 1991). LKQ introduces values in the focus of a stoup on the right ($\Gamma \vdash V : A \,;\, \Delta$) and LKT introduces co-values in the focus of a stoup on the left ($\Gamma \,;\, E : A \vdash \Delta$). Notice how the focus of the inference rules is forcibly maintained through type checking: working bottom-up, once a (co-)value is in focus in the stoup, our active attention cannot move to any other type in the sequent via activation since the *AR* and *AL* rules do not introduce (co-)values in focus. The new form of sequent calls for additional focusing structural rules *FR* (in LKQ) and

$$\begin{aligned}
A, B, C \in \mathit{Type} &::= X \mid A \times B \mid A + B \mid \neg A \mid A \to B \mid \forall X.A \mid \exists X.A \\
v \in \mathit{Term} &::= V \mid \mu\alpha.c \\
V \in \mathit{Value} &::= x \mid (V, V) \mid \iota_1\,(V) \mid \iota_2\,(V) \mid \mathsf{not}(e) \mid \lambda x.v \mid \Lambda X.v \mid A \mathbin{@} V \\
e \in \mathit{CoTerm} &::= \alpha \mid \tilde{\mu}x.c \mid \pi_1\,[e] \mid \pi_2\,[e] \mid [e, e] \mid \mathsf{not}[v] \mid v \cdot e \mid B \mathbin{@} e \mid \tilde{\Lambda}X.e \\
c \in \mathit{Command} &::= \langle v \| e \rangle \\
\mathit{Judgement} &::= c : (\Gamma \vdash \Delta) \mid (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \vdash V : A \,;\, \Delta) \mid (\Gamma \mid e : A \vdash \Delta)
\end{aligned}$$

Axiom:

$$\dfrac{}{x : A \vdash x : A \,;}\ \mathit{Var} \qquad\qquad \dfrac{}{\mid \alpha : A \vdash \alpha : A}\ \mathit{CoVar}$$

Logical rules:

$$\dfrac{\Gamma \vdash V : A \,;\, \Delta \quad \Gamma \vdash V' : B \,;\, \Delta}{\Gamma \vdash (V, V') : A \times B \,;\, \Delta}\ \times R$$

$$\dfrac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \pi_1[e] : A \times B \vdash \Delta}\ \times L_1 \qquad \dfrac{\Gamma \mid e : B \vdash \Delta}{\Gamma \mid \pi_2[e] : A \times B \vdash \Delta}\ \times L_2$$

$$\dfrac{\Gamma \vdash V : A \,;\, \Delta}{\Gamma \vdash \iota_1(V) : A + B \,;\, \Delta}\ +R_1 \qquad \dfrac{\Gamma \vdash V : B \,;\, \Delta}{\Gamma \vdash \iota_2(V) : A + B \,;\, \Delta}\ +R_2$$

$$\dfrac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid e' : B \vdash \Delta}{\Gamma \mid [e, e'] : A + B \vdash \Delta}\ +L$$

$$\dfrac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash \mathsf{not}(e) : \neg A \,;\, \Delta}\ \neg R \qquad \dfrac{\Gamma \vdash v : A \mid \Delta}{\Gamma \mid \mathsf{not}[v] : \neg A \vdash \Delta}\ \neg L$$

$$\dfrac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \,;\, \Delta}\ \to R \qquad \dfrac{\Gamma \vdash V : A \,;\, \Delta \quad \Gamma' \mid e : B \vdash \Delta'}{\Gamma, \Gamma' \mid V \cdot e : A \to B \vdash \Delta, \Delta'}\ \to L$$

$$\dfrac{\Gamma \vdash v : A \mid \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \Lambda X.v : \forall X.A \,;\, \Delta}\ \forall R \qquad \dfrac{\Gamma \mid e : A\{B/X\} \vdash \Delta}{\Gamma \mid B \mathbin{@} e : \forall X.A \vdash \Delta}\ \forall L$$

$$\dfrac{\Gamma \vdash V : A\{B/X\} \,;\, \Delta}{\Gamma \vdash B \mathbin{@} V : \exists X.A \,;\, \Delta}\ \exists R \qquad \dfrac{\Gamma \mid e : A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\Lambda}X.e : \exists X.A \vdash \Delta}\ \exists L$$

Focusing (structural) rules:

$$\dfrac{\Gamma \vdash V : A \,;\, \Delta}{\Gamma \vdash V : A \mid \Delta}\ FR$$

FIGURE 3.15. LKQ: The focused sub-syntax and types for the call-by-value dual calculus.

$$\begin{aligned}
A, B, C \in \mathit{Type} &::= X \mid A \times B \mid A + B \mid \neg A \mid A \to B \mid \forall X.A \mid \exists X.A \\
v \in \mathit{Term} &::= x \mid \mu\alpha.c \mid (v, v) \mid \iota_1\,(v) \mid \iota_2\,(v) \mid \mathsf{not}(e) \mid \lambda x.v \mid \Lambda X.v \mid B \mathbin{@} v \\
e \in \mathit{CoTerm} &::= E \mid \tilde{\mu}x.c \\
E \in \mathit{CoValue} &::= \alpha \mid \pi_1\,[E] \mid \pi_2\,[E] \mid [E, E] \mid \mathsf{not}(v) \mid v \cdot E \mid B \mathbin{@} E \mid \tilde{\Lambda}X.e \\
c \in \mathit{Command} &::= \langle v \| e \rangle \\
\mathit{Sequent} &::= c : (\Gamma \vdash \Delta) \mid (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta) \mid (\Gamma \,;\, E : A \vdash \Delta)
\end{aligned}$$

Axiom:

$$\frac{}{x : A \vdash x : A \mid}\ \mathit{Var} \qquad \frac{}{;\, \alpha : A \vdash \alpha : A}\ \mathit{CoVar}$$

Logical rules:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash v' : B \mid \Delta}{\Gamma \vdash (v, v') : A \times B \mid \Delta}\ \times R$$

$$\frac{\Gamma \,;\, E : A \vdash \Delta}{\Gamma \,;\, \pi_1[E] : A \times B \vdash \Delta}\ \times L_1 \qquad \frac{\Gamma \,;\, E : B \vdash \Delta}{\Gamma \,;\, \pi_2[E] : A \times B \vdash \Delta}\ \times L_2$$

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_1(v) : A + B \mid \Delta}\ +R_1 \qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \iota_2(v) : A + B \mid \Delta}\ +R_2$$

$$\frac{\Gamma \,;\, e : A \vdash \Delta \quad \Gamma \,;\, e' : B \vdash \Delta}{\Gamma \,;\, [E, E'] : A + B \vdash \Delta}\ +L$$

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash \mathsf{not}(e) : \neg A \mid \Delta}\ \neg R \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \,;\, \mathsf{not}[v] : \neg A \vdash \Delta}\ \neg L$$

$$\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta}\ \to R \qquad \frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \,;\, E : B \vdash \Delta'}{\Gamma, \Gamma' \,;\, v \cdot E : A \to B \vdash \Delta, \Delta'}\ \to L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \Lambda X.v : \forall X.A \mid \Delta}\ \forall R \qquad \frac{\Gamma \,;\, E : A\,\{B/X\} \vdash \Delta}{\Gamma \,;\, B \mathbin{@} E : \forall X.A \vdash \Delta}\ \forall L$$

$$\frac{\Gamma \vdash v : A\,\{B/X\} \vdash \Delta}{\Gamma \vdash B \mathbin{@} v : \exists X.A \mid \Delta}\ \exists R \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \,;\, \tilde{\Lambda}X.e : \exists X.A \vdash \Delta}\ \exists L$$

Focusing (structural) rules:

$$\frac{\Gamma \,;\, E : A \vdash \Delta}{\Gamma \mid E : A \vdash \Delta}\ FL$$

FIGURE 3.16. LKT: The focused sub-syntax and types for the call-by-name dual calculus.

FL (in LKT) which just say that every value is a term and every co-value is a co-term. However, the reverse of the focusing rules—which would say that every (co-)term is a (co-)value—are omitted in LKQ and LKT because they would collapse the distinction that the stoup has created. As it turns out (Curien & Munch-Maccagnoni, 2010), distinguishing (co-)values in type systems like LKQ and LKT correspond with the technique of focusing in proof theory developed by Andreoli (1992), Girard (1993, 2001), and Laurent (2002). In proof search, focusing makes the searching algorithm more efficient by cutting down on the search space, whereas in calculi, focusing identifies a well-behaved sub-syntax for the operational semantics.

Dynamic focusing

For the dynamic method of focusing, consider which steps were missing from the operational semantics. So instead of ruling out troublesome corners of the syntax, we will instead add additional steps to kick-start stuck commands. Recall that in our stuck call-by-value command, $\langle((\mu\beta.\langle f\|1\cdot\beta\rangle),2)\|\pi_1[\alpha]\rangle$, the $\beta^{\times}_{\mathcal{V}}$ operational step was stuck because a pair with a non-value component needs to interact with a projection. One thing we can do in this situation is lift the non-value component out of the pair and assign it a name via an input abstraction. Such a step reveals a hidden $\mu_{\mathcal{V}}$ reduction and lets the computation continue to bring the application of f to the top:

$$\begin{aligned}\langle((\mu\beta.\langle f\|1\cdot\beta\rangle),2)\|\pi_1[\alpha]\rangle &\mapsto_{?} \langle\mu\beta.\langle f\|1\cdot\beta\rangle\|\tilde{\mu}x.\langle(x,2)\|\pi_1[\alpha]\rangle\rangle\\ &\mapsto_{\mu_{\mathcal{V}}} \langle f\|1\cdot\tilde{\mu}x.\langle(x,2)\|\pi_1[\alpha]\rangle\rangle\end{aligned}$$

Now, assuming that the call to f returns the result 3, the computation can continue along to present 3 as the result to α, yielding the desired answer:

$$\begin{aligned}\langle f\|1\cdot\tilde{\mu}x.\langle(x,2)\|\pi_1[\alpha]\rangle\rangle &\mapsto\!\!\!\!\rightarrow \langle 3\|\tilde{\mu}x.\langle(x,2)\|\pi_1[\alpha]\rangle\rangle\\ &\mapsto_{\tilde{\mu}_{\mathcal{V}}} \langle(3,2)\|\pi_1[\alpha]\rangle\\ &\mapsto_{\beta^{\times}_{\mathcal{V}}} \langle 3\|\alpha\rangle\end{aligned}$$

That one extra lifting step was all that was needed to continue the computation and get to the final command. Likewise, the stuck call-by-name command $\langle(1,2)\|\pi_1[\tilde{\mu}x.\langle 0\|\alpha\rangle]\rangle$ has a non-co-value component in the projection, so we can similarly lift the component out of the projection and assign it a name via an output abstraction:

$$\begin{aligned}\langle (1,2) \| \pi_1 \left[\tilde{\mu} x.\, \langle 0 \| \alpha \rangle\right] \rangle &\mapsto_{?} \langle \mu \beta.\, \langle (1,2) \| \pi_1 \left[\beta\right] \rangle \| \tilde{\mu} x.\, \langle 0 \| \alpha \rangle \rangle \\ &\mapsto_{\tilde{\mu}_{\mathcal{N}}} \langle 0 \| \alpha \rangle\end{aligned}$$

Lifting non-(co-)value components out of introduction forms of (co-)terms seems to be the missing step in β-stuck commands.

The full set of such lifting rules are given in Figure 3.17 for the call-by-value semantics and Figure 3.18 for the call-by-name semantics.[8] These rules give the minimum required extra steps to reduce hidden computations nested deeply inside terms and co-terms in a way that matches the call-by-value and call-by-name semantics for the λ-calculus. However, the ς laws are the first operational rules on (co-)terms, rather than commands. As such, we must extend the context of our operational reductions to allow for ς when necessary. For the call-by-value and call-by-name operational semantics including ς, we have the following evaluation contexts (denoted by D to avoid confusion with co-values):

$$D \in \mathit{EvalCxt}_{\mathcal{V}} ::= \Box \mid \langle \Box \| e \rangle \mid \langle V \| \Box \rangle \qquad D \in \mathit{EvalCxt}_{\mathcal{N}} ::= \Box \mid \langle v \| \Box \rangle \mid \langle \Box \| E \rangle$$

Still, unlike the λ-context, evaluation contexts are not arbitrarily nested, but only ever place attention the entire command or its immediate (co-)term. For example, in call-by-value we have the following operational ς reductions on either side of a command like:

$$\begin{aligned}\langle \iota_1 (v) \| e \rangle &\mapsto_{\varsigma^{+}_{\mathcal{V}}} \langle \mu \alpha.\, \langle v \| \tilde{\mu} y.\, \langle \iota_1 (y) \| \alpha \rangle \rangle \| e \rangle \mapsto_{\mu_{\mathcal{V}}} \langle v \| \tilde{\mu} y.\, \langle \iota_1 (y) \| e \rangle \rangle \\ \langle V \| v \cdot e \rangle &\mapsto_{\varsigma^{\rightarrow}_{\mathcal{N}}} \langle V \| \tilde{\mu} x.\, \langle v \| \tilde{\mu} y.\, \langle x \| y \cdot e \rangle \rangle \rangle \mapsto_{\tilde{\mu}_{\mathcal{V}}} \langle v \| \tilde{\mu} y.\, \langle V \| y \cdot e \rangle \rangle\end{aligned}$$

and in call-by-name we have only operational ς reductions on the co-term side like:

$$\langle v \| \pi_1 \left[e\right] \rangle \mapsto_{\varsigma^{\times}_{\mathcal{N}}} \langle v \| \tilde{\mu} x.\, \langle \mu \beta.\, \langle x \| \pi_1 \left[\beta\right] \rangle \| e \rangle \rangle \mapsto_{\tilde{\mu}_{\mathcal{N}}} \langle \mu \beta.\, \langle v \| \pi_1 \left[\beta\right] \rangle \| e \rangle$$

Furthermore, note that extending the semantics of the dual calculi with the ς rules preserves determinism of the operational semantics and confluence of the reduction

[8] The proviso that x, y, α, and β are *fresh* means that they do not appear free anywhere in the command on the left-hand side of the operational reduction step.

$$\left.\begin{array}{lrl}
(\varsigma_{\mathcal{V}}^{\times}) & (v, v') \succ_{\varsigma_{\mathcal{V}}^{\times}} \mu\alpha.\, \langle v \| \tilde{\mu} y.\, \langle (y, v') \| \alpha \rangle \rangle & (V, v) \succ_{\varsigma_{\mathcal{V}}^{\times}} \mu\alpha.\, \langle v \| \tilde{\mu} y.\, \langle (V, y) \| \alpha \rangle \rangle \\
(\varsigma_{\mathcal{V}}^{+}) & \iota_i\,(v) \succ_{\varsigma_{\mathcal{V}}^{+}} \mu\alpha.\, \langle v \| \tilde{\mu} y.\, \langle \iota_i\,(y) \| \alpha \rangle \rangle & \\
(\varsigma_{\mathcal{V}}^{\rightarrow}) & v \cdot e \succ_{\varsigma_{\mathcal{V}}^{\rightarrow}} \tilde{\mu} x.\, \langle v \| \tilde{\mu} y.\, \langle x \| y \cdot e \rangle \rangle & \\
(\varsigma_{\mathcal{V}}^{\exists}) & B \mathbin{@} v \succ_{\varsigma_{\mathcal{V}}^{\exists}} \mu\alpha.\, \langle v \| \tilde{\mu} y.\, \langle B \mathbin{@} y \| \alpha \rangle \rangle &
\end{array}\right\} \begin{array}{l} v \notin \mathit{Value}_{\mathcal{V}} \\ \alpha, x, y \text{ fresh} \end{array}$$

FIGURE 3.17. The focusing ς laws for the call-by-value ($\mathcal{V}$) half of the dual calculi.

$$\left.\begin{array}{lrl}
(\varsigma_{\mathcal{N}}^{\times}) & \pi_i\,[e] \succ_{\varsigma_{\mathcal{N}}^{\times}} \tilde{\mu} x.\, \langle \mu\beta.\, \langle x \| \pi_i\,[\beta] \rangle \| e \rangle & \\
(\varsigma_{\mathcal{N}}^{+}) & [e, e'] \succ_{\varsigma_{\mathcal{N}}^{+}} \tilde{\mu} x.\, \langle \mu\beta.\, \langle x \| [\beta, e'] \rangle \| e \rangle & [E, e] \succ_{\varsigma_{\mathcal{N}}^{+}} \tilde{\mu} x.\, \langle \mu\beta.\, \langle x \| [E, \beta] \rangle \| e \rangle \\
(\varsigma_{\mathcal{N}}^{\rightarrow}) & v \cdot e \succ_{\varsigma_{\mathcal{N}}^{\rightarrow}} \tilde{\mu} x.\, \langle \mu\beta.\, \langle x \| v \cdot \beta \rangle \| e \rangle & \\
(\varsigma_{\mathcal{N}}^{\forall}) & B \mathbin{@} e \succ_{\varsigma_{\mathcal{N}}^{\forall}} \tilde{\mu} x.\, \langle \mu\beta.\, \langle x \| B \mathbin{@} \beta \rangle \| e \rangle &
\end{array}\right\} \begin{array}{l} e \notin \mathit{CoValue}_{\mathcal{N}} \\ x, \beta \text{ fresh} \end{array}$$

FIGURE 3.18. The focusing ς laws for the call-by-name ($\mathcal{N}$) half of the dual calculi.

theory, since there are no critical pairs between the ς rules and $\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}\beta$ rules in either the call-by-value or call-by-name calculus.

For the $\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}\varsigma_{\mathcal{V}}$ call-by-value operational semantics, the net effect is that the final commands are always a *value* yielded to a co-variable or a *simple co-value* (that is, a co-variable or a left introduction co-term) applied to a variable as follows:

$$\begin{aligned}
\mathit{FinalCommand}_{\mathcal{V}} &::= \langle V \| \alpha \rangle \mid \langle x \| E_s \rangle \\
V \in \mathit{Value}_{\mathcal{V}} &::= x \mid (V, V') \mid \iota_1\,(V) \mid \iota_2\,(V) \mid \mathsf{not}(e) \mid \lambda x.v \mid \Lambda X.v \mid B \mathbin{@} V \\
E_s \in \mathit{SimpleCoValue}_{\mathcal{V}} &::= \alpha \mid \pi_1\,[e] \mid \pi_2\,[e] \mid [e, e'] \mid \mathsf{not}[v] \mid V \cdot e \mid B \mathbin{@} e \mid \tilde{\Lambda} X.e
\end{aligned}$$

Dually for the $\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}\beta_{\mathcal{N}}\varsigma_{\mathcal{N}}$ call-by-name operational semantics, the final commands are always a *simple value* (a variable or an introduction term) yielded to a co-variable or a *co-value* applied to a variable as follows:

$$\begin{aligned}
\mathit{FinalCommand}_{\mathcal{N}} &::= \langle V_s \| \alpha \rangle \mid \langle x \| E \rangle \\
V_s \in \mathit{SimpleValue}_{\mathcal{N}} &::= x \mid (v, v') \mid \iota_1\,(v) \mid \iota_2\,(v) \mid \mathsf{not}(e) \mid \lambda x.v \mid \Lambda X.v \mid B \mathbin{@} v \\
E \in \mathit{CoValue}_{\mathcal{N}} &::= \alpha \mid \pi_1\,[E] \mid \pi_2\,[E] \mid [E, E'] \mid \mathsf{not}[v] \mid v \cdot E \mid B \mathbin{@} E \mid \tilde{\Lambda} X.e
\end{aligned}$$

If we only take well-typed commands into consideration, then we get a standard type safety theorem which says that well-typed commands always reduce to a final command, and do not get stuck on any interacting (and potentially mismatched) introduction forms. The small-step version of type safety can be expressed as the *progress* and *preservation* properties (Wright & Felleisen, 1994).

Theorem 3.3 (Progress and preservation)**.** *For any command $c : (\Gamma \vdash \Delta)$:*

a) Progress: *c is a call-by-value (respectively, call-by-name) final command or there is a command c' such that $c \mapsto_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}\varsigma_{\mathcal{V}}} c'$ (respectively, $c \mapsto_{\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}\beta_{\mathcal{N}}\varsigma_{\mathcal{N}}} c'$), and*

b) Preservation: *if $c \mapsto_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}\varsigma_{\mathcal{V}}} c'$ or $c \mapsto_{\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}\beta_{\mathcal{N}}\varsigma_{\mathcal{N}}} c'$, then $c' : (\Gamma \vdash \Delta)$.*

Proof. Progress follows by induction on the typing derivation of $c : (\Gamma \vdash \Delta)$. The structural rules (for weakening, contraction, and exchange) follow immediately from the inductive hypothesis and the *Cut* rule forms the base cases. For call-by-name, progress is assured because for every well-typed co-term $\Gamma \mid e : A \vdash \Delta$, either e is a co-value, an input abstraction, or $e \succ_{\varsigma_{\mathcal{N}}} e'$ for some e'. Therefore, if the cut is neither final nor reducible, then the co-term reduces. Similarly for call-by-value, every well-typed term $\Gamma \vdash v : A \mid \Delta$ is either a value, an output abstraction or $v \succ_{\varsigma_{\mathcal{V}}} v'$ for some v', and every well-typed co-term $\Gamma \mid e : A \vdash \Delta$ is either a simple co-value, an input abstraction, or $e \succ_{\varsigma_{\mathcal{V}}} e'$ for some e'. Therefore, if the cut is neither final nor reducible, then either the term reduces or the term is a value and the co-term reduces.

Preservation follows by cases on all the possible rewriting rules so that

– if $c \succ_{\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}\beta\varsigma} c'$ then $c : (\Gamma \vdash \Delta)$ implies $c' : (\Gamma \vdash \Delta)$,

– if $v \succ_{\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}\beta\varsigma} v'$ then $v : (\Gamma \vdash \Delta)\, C$ implies $v' : (\Gamma \vdash \Delta)\, C$, and

– if $e \succ_{\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}\beta\varsigma} e'$ then $e : (\Gamma \vdash \Delta)\, C$ implies $e' : (\Gamma \vdash \Delta)\, C$.

for both call-by-value and call-by-name, using the fact that for $\Gamma \vdash V : A \mid \Delta$ and $\Gamma' \mid E : A \vdash \Delta'$:

– if $c : (\Gamma', x : A \vdash \Delta')$ then $c\{V/x\} : (\Gamma', \Gamma \vdash \Delta', \Delta)$,

– if $c : (\Gamma \vdash \alpha : A, \Delta)$ then $c\{E/\alpha\} : (\Gamma', \Gamma \vdash \Delta', \Delta)$,

– if $\Gamma', x : A \vdash v : C \mid \Delta'$ then $\Gamma', \Gamma \vdash v\,\{V/x\} : C \mid \Delta', \Delta$,

- if $\Gamma \vdash v : C \mid \alpha : A, \Delta$ then $\Gamma', \Gamma \vdash v\,\{E/\alpha\} : C \mid \Delta', \Delta$,
- if $\Gamma \vdash v : C \mid \Delta$ and $X \notin FV(\Gamma \vdash \Delta)$ then $\Gamma \vdash v\,\{B/X\} : C\,\{B/X\} \mid \Delta$,
- if $\Gamma', x : A \mid e : C \vdash \Delta'$ then $\Gamma', \Gamma \mid e\,\{V/x\} : C \vdash \Delta', \Delta$,
- if $\Gamma \mid e : C \vdash \alpha : A, \Delta$ then $\Gamma', \Gamma \mid e\,\{E/\alpha\} : C \vdash \Delta', \Delta$, and
- if $\Gamma \mid e : C \vdash \Delta$ and $X \notin FV(\Gamma \vdash \Delta)$ then $\Gamma \mid e\,\{B/X\} : C\,\{B/X\} \vdash \Delta$,

each of which follows by induction on the typing derivation of c, $v : C$ and $e : C$. □

From progress and preservation, we can derive the following big-step statement of type safety.

Theorem 3.4 (Type safety)**.** *For any dual calculi command $c : (\Gamma \vdash \Delta)$:*

- *if $c \mapsto\!\!\!\!\twoheadrightarrow_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}\varsigma_{\mathcal{V}}} c'$ then $c' : (\Gamma \vdash \Delta)$ and c' is irreducible (i.e. $c' \not\mapsto_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}\varsigma_{\mathcal{V}}}$) if and only if c' is a call-by-value final command, and*
- *if $c \mapsto\!\!\!\!\twoheadrightarrow_{\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}\beta_{\mathcal{N}}\varsigma_{\mathcal{N}}} c'$, then $c' : (\Gamma \vdash \Delta)$ and c' is irreducible (i.e. $c' \not\mapsto_{\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}\beta_{\mathcal{N}}\varsigma_{\mathcal{N}}}$) if and only if c' is a call-by-name final command.*

Proof. By induction on the left-to-right reflexive-transitive structure of $c \mapsto\!\!\!\!\twoheadrightarrow_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}\varsigma_{\mathcal{V}}} c'$ and $c \mapsto\!\!\!\!\twoheadrightarrow_{\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}\beta_{\mathcal{N}}\varsigma_{\mathcal{N}}} c'$, using progress (Theorem 3.3 (a)) for the reflexive case and preservation (Theorem 3.3 (b)) for the transitive case. □

Remark 3.8. The original $\overline{\lambda}\mu\tilde{\mu}$-calculus used a different β rule for functions, namely:

$$(\beta^{\to}) \qquad \langle \lambda x.v \| v' \cdot e \rangle \succ_{\beta^{\to}} \langle v' \| \tilde{\mu}x.\, \langle v \| e \rangle \rangle \qquad x \notin FV(e)$$

This $\beta^{\to}$ works the same for both call-by-name and call-by-value reduction; since the argument v' is bound to x with an input abstraction, the rules of the core $\mu\tilde{\mu}$-calculus take over to determine whether or not the argument is evaluated now (by a $\mu_{\mathcal{V}}$ reduction, for example) or later (by a $\tilde{\mu}_{\mathcal{N}}$ reduction). Furthermore, this form of $\beta^{\to}$ reduction applies more often than the strategy-specific $\beta^{\to}_{\mathcal{V}}$ and $\beta^{\to}_{\mathcal{N}}$, so we might ask if it avoids the need of focusing for functions altogether. Unfortunately, the general $\beta^{\to}$ rule still suffers a similar, if more subtle, fate as the strategy-specific β rules. For example, consider the command $\langle f \| \mu\beta.\, \langle 1 \| \alpha \rangle \cdot \tilde{\mu}x.\, \langle 0 \| \alpha \rangle \rangle$ which corresponds to the expression $\mathbf{let}\, z = f\ (\mathsf{abort}\, 1)\, \mathbf{in}\, 0$ in a functional language containing the control operator abort that halts the current computation and yields its argument as the result. In call-by-value this expression should evaluate to 1, and in call-by-name it should evaluate to 0, but the $\beta^{\to}$ rule does not help us since there is a free variable f instead of a λ-abstraction. In this command, the ς rules are still necessary to get the final result, and unfortunately combining the general $\beta^{\to}$ rule with $\varsigma^{\to}$ creates a mild form of non-determinism in the operational semantics since some $\beta^{\to}$ redexes are also $\varsigma^{\to}$ redexes (though the associated reduction theories are still confluent).

As it turns out, though, the combination of lifting and strategy-specific $\beta^{\to}$ reductions are more powerful than the generalized $\beta^{\to}$ rule. In call-by-value, the combination of $\varsigma_{\mathcal{V}}^{\to}$, $\tilde{\mu}_{\mathcal{V}}$, and $\beta_{\mathcal{V}}^{\to}$ exactly simulate the $\bar{\lambda}\mu\tilde{\mu}$-calculus $\beta^{\to}$ rule as follows:

$$\langle \lambda x.v \| v' \cdot e \rangle \mapsto_{\varsigma_{\mathcal{V}}^{\to}} \langle \lambda x.v \| \tilde{\mu} y.\, \langle v' \| \tilde{\mu} x.\, \langle y \| x \cdot e \rangle \rangle \rangle \mapsto_{\tilde{\mu}_{\mathcal{V}}} \langle v' \| \tilde{\mu} x.\, \langle \lambda x.v \| x \cdot e \rangle \rangle \to_{\beta_{\mathcal{V}}^{\to}} \langle v \| \tilde{\mu} x.\, \langle v \| e \rangle \rangle$$

In call-by-name, observe that the combination of $\bar{\lambda}\mu\tilde{\mu}$'s $\beta^{\to}$ and $\tilde{\mu}_{\mathcal{N}}$ rules simulate the call-by-name-specific $\beta_{\mathcal{N}}^{\to}$ even when the call stack is not a co-value,

$$\langle \lambda x.v \| v' \cdot e \rangle \mapsto_{\beta^{\to}} \langle v' \| \tilde{\mu} x.\, \langle v \| e \rangle \rangle \mapsto_{\tilde{\mu}_{\mathcal{N}}} \langle v \left\{ v'/x \right\} \| e \rangle$$

but together the $\tilde{\mu}_{\mathcal{N}} \eta_{\mu} \beta_{\mathcal{N}}^{\to} \varsigma_{\mathcal{N}}^{\to}$ rules perform the same reduction as follows:

$$\begin{aligned} \langle \lambda x.v \| v' \cdot e \rangle &\mapsto_{\varsigma_{\mathcal{N}}^{\to}} \langle \lambda x.v \| \tilde{\mu} y.\, \langle \mu \alpha.\, \langle y \| v' \cdot \alpha \rangle \| e \rangle \rangle \mapsto_{\tilde{\mu}_{\mathcal{N}}} \langle \mu \alpha.\, \langle \lambda x.v \| v' \cdot \alpha \rangle \| e \rangle \\ &\to_{\beta_{\mathcal{N}}^{\to}} \langle \mu \alpha.\, \langle v \left\{ v'/x \right\} \| \alpha \rangle \| e \rangle \to_{\eta_{\mu}} \langle v \left\{ v'/x \right\} \| e \rangle \end{aligned}$$

So even though type safety (Theorem 3.4) cannot dispense with the $\varsigma^{\to}$ rules by adopting the $\bar{\lambda}\mu\tilde{\mu}$-calculus' original $\beta^{\to}$ rules, we can still rely on the combination of strategy-specific $\beta^{\to}\varsigma^{\to}$ rules from Figures 3.13 and 3.17 and Figures 3.14 and 3.18 to get all the same results with deterministic operational semantics. *End remark* 3.8.

Static versus dynamic focusing

Now that we have two different methods for addressing β-stuck commands, one question still remains: what do the static and dynamic methods have to do with one another? As it turns out, they are compatible and complementary solutions to the same problem—two sides of the same coin—that apply the same essential idea at different times. First, one of the major features of static focusing in proof theories and type systems is that the apparent restriction on inference rules is no real restriction at

$$\llbracket \langle v \| e \rangle \rrbracket^Q \triangleq \left\langle \llbracket v \rrbracket^Q \middle\| \llbracket e \rrbracket^Q \right\rangle$$

$$\begin{array}{rlrl}
\llbracket x \rrbracket^Q & \triangleq x & \llbracket \mu\alpha.c \rrbracket^Q & \triangleq \mu\alpha.\llbracket c \rrbracket^Q \\
\llbracket (v, v') \rrbracket^Q & \triangleq \mu\alpha.\left\langle \llbracket v \rrbracket^Q \middle\| \tilde{\mu}x.\left\langle \llbracket (x, v') \rrbracket^Q \middle\| \alpha \right\rangle \right\rangle & \llbracket (V, v) \rrbracket^Q & \triangleq \mu\alpha.\left\langle \llbracket v \rrbracket^Q \middle\| \tilde{\mu}x.\left\langle \llbracket (V, x) \rrbracket^Q \middle\| \alpha \right\rangle \right\rangle \\
\llbracket (V, V') \rrbracket^Q & \triangleq \left(\llbracket V \rrbracket^Q, \llbracket V' \rrbracket^Q \right) & & \\
\llbracket \iota_i\,(v) \rrbracket^Q & \triangleq \mu\alpha.\left\langle \llbracket v \rrbracket^Q \middle\| \tilde{\mu}x.\left\langle \llbracket \iota_i\,(x) \rrbracket^Q \middle\| \alpha \right\rangle \right\rangle & \llbracket \iota_i\,(V) \rrbracket^Q & \triangleq \iota_i \left(\llbracket V \rrbracket^Q \right) \\
\llbracket \mathsf{not}(e) \rrbracket^Q & \triangleq \mathsf{not}(\llbracket e \rrbracket^Q) & & \\
\llbracket \lambda x.v' \rrbracket^Q & \triangleq \lambda x.\llbracket v' \rrbracket^Q & \llbracket \Lambda X.v' \rrbracket^Q & \triangleq \Lambda X.\llbracket v' \rrbracket^Q \\
\llbracket B \mathbin{@} v \rrbracket^Q & \triangleq \mu\alpha.\left\langle v \middle\| \tilde{\mu}x.\left\langle \llbracket B \mathbin{@} x \rrbracket^Q \middle\| \alpha \right\rangle \right\rangle & \llbracket B \mathbin{@} V \rrbracket^Q & \triangleq B \mathbin{@} \llbracket V \rrbracket^Q
\end{array}$$

$$\begin{array}{rlrlrl}
\llbracket \alpha \rrbracket^Q & \triangleq \alpha & \llbracket \tilde{\mu}x.c \rrbracket^Q & \triangleq \tilde{\mu}x.\llbracket c \rrbracket^Q & & \\
\llbracket \pi_i\,[e] \rrbracket^Q & \triangleq \pi_i \left[\llbracket e \rrbracket^Q \right] & \llbracket [e, e'] \rrbracket^Q & \triangleq \left[\llbracket e \rrbracket^Q, \llbracket e' \rrbracket^Q \right] & \llbracket \mathsf{not}[v'] \rrbracket^Q & \triangleq \mathsf{not}[\llbracket v' \rrbracket^Q] \\
\llbracket v \cdot e \rrbracket^Q & \triangleq \tilde{\mu}x.\left\langle \llbracket v \rrbracket^Q \middle\| \tilde{\mu}y.\left\langle x \middle\| \llbracket y \cdot e \rrbracket^Q \right\rangle \right\rangle & \llbracket V \cdot e \rrbracket^Q & \triangleq \llbracket V \rrbracket^Q \cdot \llbracket e \rrbracket^Q & & \\
\llbracket B \mathbin{@} e \rrbracket^Q & \triangleq B \mathbin{@} \llbracket e \rrbracket^Q & \left\llbracket \tilde{\Lambda}X.e \right\rrbracket^Q & \triangleq \tilde{\Lambda}X.\llbracket e \rrbracket^Q & &
\end{array}$$

$$\textbf{where}\, v \notin \mathit{Value}_{\mathcal{V}}$$

FIGURE 3.19. The Q-focusing translation to the LKQ sub-syntax.

all: every program (i.e. proof) in the original system has a corresponding program with the same type (i.e. specification) in the focused sub-system. We can make this claim more formally for LKQ and LKT by observing that the syntactic transformations in Figures 3.19 and 3.20 translate general dual calculi expressions into the LKQ and LKT sub-syntaxes, respectively, with the same type (by generalizing the proof of preservation in Theorem 3.3 (b)). These translations are defined in such a way that an expression that happens to already lie in the LKQ sub-syntax is not altered by Q-focusing translation, and likewise LKT expressions are not altered by T-focusing translation.

With the focusing translations and the ς reduction theory in hand, we can now observe that both the static and dynamic methods of focusing amount to the same thing. In particular, notice that the LKQ sub-syntax is just the $\varsigma_{\mathcal{V}}$-normal forms from the original dual calculus and the Q-focusing translation performs call-by-value $\varsigma_{\mathcal{V}}$-normalization, and similarly the T-focusing translation is just call-by-name $\varsigma_{\mathcal{N}}$-

$$\llbracket \langle v \| e \rangle \rrbracket^T \triangleq \left\langle \llbracket v \rrbracket^T \middle\| \llbracket e \rrbracket^T \right\rangle$$

$$\begin{array}{lll}
\llbracket x \rrbracket^T \triangleq x & \llbracket \mu\alpha.c \rrbracket^T \triangleq \mu\alpha.\llbracket c \rrbracket^T & \\
\llbracket (v, v') \rrbracket^T \triangleq \left(\llbracket v \rrbracket^T, \llbracket v' \rrbracket^T \right) & \llbracket \iota_i \, (v) \rrbracket^T \triangleq \pi_i \left[\llbracket v \rrbracket^T \right] & \llbracket \mathsf{not}(e) \rrbracket^T \triangleq \mathsf{not}(\llbracket e \rrbracket^T) \\
\llbracket \lambda x.v \rrbracket^T \triangleq \lambda x.\llbracket v \rrbracket^T & \llbracket \Lambda X.v \rrbracket^T \triangleq \Lambda X.\llbracket v \rrbracket^T & \llbracket B \mathbin{@} v \rrbracket^T \triangleq B \mathbin{@} \llbracket v \rrbracket^T
\end{array}$$

$$\begin{array}{ll}
\llbracket \alpha \rrbracket^T \triangleq \alpha & \llbracket \tilde{\mu} x.c \rrbracket^T \triangleq \tilde{\mu} x.\llbracket c \rrbracket^T \\
\llbracket \pi_i \, [e] \rrbracket^T \triangleq \tilde{\mu} x. \left\langle \mu\alpha. \left\langle x \middle\| \llbracket \pi_i \, [\alpha] \rrbracket^T \right\rangle \middle\| \llbracket e \rrbracket^T \right\rangle & \llbracket \pi_i \, [E] \rrbracket^T \triangleq \pi_i \left[\llbracket E \rrbracket^T \right] \\
\llbracket [e, e'] \rrbracket^T \triangleq \tilde{\mu} x. \left\langle \mu\alpha. \left\langle x \middle\| \llbracket [\alpha, e'] \rrbracket^T \right\rangle \middle\| \llbracket e \rrbracket^T \right\rangle & \llbracket [E, e] \rrbracket^T \triangleq \tilde{\mu} x. \left\langle \mu\alpha. \left\langle x \middle\| \llbracket [E, \alpha] \rrbracket^T \right\rangle \middle\| \llbracket e \rrbracket^T \right\rangle \\
\llbracket [E, E'] \rrbracket^T \triangleq \left[\llbracket E \rrbracket^T, \llbracket E' \rrbracket^T \right] & \llbracket \mathsf{not}[v] \rrbracket^T \triangleq \mathsf{not}[\llbracket v \rrbracket^T] \\
\llbracket v \cdot e \rrbracket^T \triangleq \tilde{\mu} x. \left\langle \mu\alpha. \left\langle x \middle\| \llbracket v \cdot \alpha \rrbracket^T \right\rangle \middle\| \llbracket e \rrbracket^T \right\rangle & \llbracket v \cdot E \rrbracket^T \triangleq \llbracket v \rrbracket^T \cdot \llbracket E \rrbracket^T \\
\llbracket B \mathbin{@} e \rrbracket^T \triangleq \tilde{\mu} x. \left\langle \mu\alpha. \left\langle x \middle\| \llbracket B \mathbin{@} \alpha \rrbracket^T \right\rangle \middle\| \llbracket e \rrbracket^T \right\rangle & \llbracket B \mathbin{@} E \rrbracket^T \triangleq B \mathbin{@} \llbracket E \rrbracket^T \\
\left\llbracket \tilde{\Lambda} X.e' \right\rrbracket^T \triangleq \tilde{\Lambda} X.\llbracket e' \rrbracket^T &
\end{array}$$

$$\textbf{where}\, e \notin CoValue_{\mathcal{N}}$$

FIGURE 3.20. The T-focusing translation to the LKT sub-syntax.

normalization into the LKT sub-syntax of $\varsigma_{\mathcal{N}}$-normal forms, which can be confirmed by induction on the syntax of (co-)terms and commands.

Theorem 3.5 (Focusing). – *Every LKQ command, term, and co-term is a $\varsigma_{\mathcal{V}}$-normal form, and $c \twoheadrightarrow_{\varsigma_{\mathcal{V}}} [\![c]\!]^{Q}$, $v \twoheadrightarrow_{\varsigma_{\mathcal{V}}} [\![v]\!]^{Q}$, and $e \twoheadrightarrow_{\varsigma_{\mathcal{V}}} [\![e]\!]^{Q}$.*

– *Every LKT command, term, and co-term is a $\varsigma_{\mathcal{N}}$-normal form, and $c \twoheadrightarrow_{\varsigma_{\mathcal{N}}} [\![c]\!]^{T}$, $v \twoheadrightarrow_{\varsigma_{\mathcal{N}}} [\![v]\!]^{T}$, and $e \twoheadrightarrow_{\varsigma_{\mathcal{N}}} [\![e]\!]^{T}$.*

Proof. The fact that LKQ expressions are $\varsigma_{\mathcal{V}}$-normal forms and LKT expressions are $\varsigma_{\mathcal{N}}$-normal forms is apparent from the syntax of LKQ and LKT. Furthermore, the fact that $c \twoheadrightarrow_{\varsigma_{\mathcal{V}}} [\![c]\!]^{Q}$, $c \twoheadrightarrow_{\varsigma_{\mathcal{N}}} [\![c]\!]^{T}$, and so on follows by mutual induction on the syntax of commands and (co-)terms. □

Therefore, the difference between the static and dynamic methods of focusing is not a matter of what but when: do we prefer to leave ς redexes to happen during execution, or would we rather reduce them all up front as a preprocessing pass?

Remark 3.9. By representing a calling context with an explicit syntactic object e, we have a direct representation of a tail-recursive interpreter (Ariola *et al.*, 2009a), which can also be seen as a form of abstract machine. In particular, we may view the syntax of the dual calculi as a more abstract representation of a CEK-style machine (Felleisen & Friedman, 1986) or a Krivine-style machine (Krivine, 2007): the control (C) is represented by a term v, the continuation (K) is represented by a co-term e, and the environment (E) is implicit and instead implemented by the capture-avoiding substitution operation. Finally, the configuration state of the machine is represented by a command c. Interestingly, though, the treatment of focusing in these machines tends to be asymmetrical depending on the evaluation strategy: call-by-value abstract machines tend to rely on dynamic focusing during execution, whereas call-by-name abstract machines tend to maintain static focusing.

For example, consider a variation on a Krivine machine with implicit substitution for call-by-name evaluation of λ-calculus terms:

$$\begin{aligned} \langle v\ v' \| E\rangle &\rightsquigarrow \langle v \| E[\Box\ v']\rangle \\ \langle \lambda x.v \| E[\Box\ v']\rangle &\rightsquigarrow \langle v\,\{v'/x\} \| E\rangle \end{aligned}$$

This machine uses two forms of evaluation context—the application of the computation in question to an argument, $E[\Box\ v']$, and the empty context, $\Box$—for finding the next

β-redex to perform. We can relate the states of this call-by-name machine to the call-by-name dual calculus by translating the evaluation contexts to co-terms. The empty context can be represented by just an arbitrary co-variable α, and the application to an argument is represented directly as a call stack co-term: $E[\Box\ v'] \triangleq v' \cdot E$. With this interpretation, the first rule of the machine states the relationship between function application in the λ-calculus and call stacks in the dual calculus, and the second rule is exactly the $\beta^{\rightarrow}_{\mathcal{N}}$ operational step. Note that if we always start with a co-value in the machine state then the first rule only ever builds co-values in the LKT sub-syntax. For example, by evaluating a term v in the "empty context" as $\langle v \| \alpha \rangle$, the co-term in the machine will always be a chain of call stacks with some number of arguments like $v_1 \cdot v_2 \cdot v_3 \cdot v_4 \cdot \alpha$. Therefore, this Krivine-style machine operates within the statically focused LKT sub-syntax.

Now consider the following variation on a CEK machine with implicit substitution for call-by-value evaluation of λ-calculus terms:

$$\begin{aligned}
\langle v\ v' \| E \rangle &\leadsto \langle v \| E[\Box\ v'] \rangle \\
\langle V \| E[\Box\ v] \rangle &\leadsto \langle v \| E[V\ \Box] \rangle \\
\langle V \| E[(\lambda x.v)\ \Box] \rangle &\leadsto \langle v\,\{V/x\} \| E \rangle
\end{aligned}$$

Compared to the call-by-name machine above, the machine uses one additional form of evaluation context—the application of a function value to the computation in question $E[V\ \Box]$—for finding the next β-redex to perform. We can extend the previous translation of evaluation contexts to co-terms so that an applied function value is represented indirectly with an input abstraction: $E[V\ \Box] \triangleq \tilde{\mu}x.\, \langle V \| x \cdot E \rangle$. With this interpretation, the first rule of the machine relates function application and call stacks as before, the second rule of the machine is a combined $\varsigma^{\rightarrow}_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}$ step,

$$\langle V \| v \cdot E \rangle \mapsto_{\varsigma^{\rightarrow}_{\mathcal{V}}} \langle V \| \tilde{\mu}x.\, \langle v \| \tilde{\mu}y.\, \langle x \| y \cdot E \rangle \rangle \rangle \mapsto_{\tilde{\mu}_{\mathcal{V}}} \langle v \| \tilde{\mu}y.\, \langle V \| y \cdot E \rangle \rangle$$

and the last rule is a combined $\tilde{\mu}_{\mathcal{V}}\beta^{\rightarrow}_{\mathcal{V}}$ step:

$$\langle V \| \tilde{\mu}y.\, \langle \lambda x.v \| y \cdot E \rangle \rangle \mapsto_{\tilde{\mu}_{\mathcal{V}}} \langle \lambda x.v \| V \cdot E \rangle \mapsto_{\beta^{\rightarrow}_{\mathcal{V}}} \langle v\,\{V/x\} \| E \rangle$$

Notice that this machine does not necessarily operate within the LKQ sub-syntax: the first rule might push a non-value computation onto a call stack. In this case, the

$\varsigma_{\mathcal{V}}^{\rightarrow}$ rule is needed to refocus the machine during execution. Of course, we could avoid the need for $\varsigma_{\mathcal{V}}^{\rightarrow}$ reduction at run-time by changing our interpretation of application to pre-$\varsigma_{\mathcal{V}}^{\rightarrow}$-normalize the call stack, as in $E[\Box\ v] \triangleq \tilde{\mu}x.\, \langle v \| \tilde{\mu}y.\, \langle x \| y \cdot E \rangle\rangle$. However, this is just a matter of taste since the two timings of focusing amount to the same thing (Theorem 3.5). *End remark* 3.9.

Call-by-value is dual to call-by-name

We now turn to the duality for which the dual calculi are named. We saw how the symmetries of the sequent calculus present a logical duality that captures De Morgan duals in Section 3.1. This duality is carried over by the Curry-Howard isomorphism and presents itself as two dualities in programming languages:

(1) a duality between the *static* semantics (types) of languages, and

(2) a duality between the *dynamic* semantics (reductions) of languages.

These dualities of programming languages were first observed by Filinski (1989) from the correspondence with duality in category theory, which was later expanded upon by Selinger (2001, 2003) in the style of natural deduction. Curien & Herbelin (2000) and Wadler (2003, 2005) brought this duality to the language of sequent calculus, and show how it is better reflected in the language as a duality of syntax corresponding to the inherent symmetries in the logic.

The static aspect of duality between types comes directly from the logical duality of the sequent calculus. Since duality spins a sequent around its turnstyle, so that assumptions are exchanged with conclusions, we also have a corresponding swap in the programming language. The dual of a term v of type A is a co-term of the dual type and vice versa, so that the term and co-term components of a command are swapped. Likewise, the duality on types lines up directly with the De Morgan duality on logical propositions. For example, since the types for pairs ($\times$) and sums ($+$) correspond to conjunction ($\wedge$) and disjunction ($\vee$), we have the same relationship with the duality operation $C^{\perp}$:

$$(A \times B)^{\perp} \triangleq (A^{\perp}) + (B^{\perp}) \qquad\qquad (A + B)^{\perp} \triangleq (A^{\perp}) \times (B^{\perp})$$

Also following the De Morgan duality, negation ($\neg$) is self-dual.

However, just like we found in Gentzen's LK sequent calculus in Section 3.1, the dual calculi presented in Figure 3.12 are missing the counterpart to functions. By analogy, we complete the duality of components in the calculus by adding the dual of functions, also referred to as subtraction, that represent a transformation on co-terms, as the counterpart to a transformation on terms. The typing rules for subtraction are the same as the logical rules for subtraction in LK, and the syntax is reversed from functions in the dual calcululi:

$$\frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma' \vdash v : B \mid \Delta'}{\Gamma, \Gamma' \vdash e \cdot v : B - A \mid \Delta, \Delta'} -R \qquad\qquad \frac{\Gamma \mid e : B \vdash \alpha : A, \Delta}{\Gamma \mid \tilde{\lambda}\alpha.e : B - A \vdash \Delta} -L$$

Similarly, the β^- and ς^- operational rules for subtraction are the mirror image of the corresponding rules for functions. In call-by-value we have:

$$\begin{array}{lrl} & Value_{\mathcal{V}} ::= & \dots \mid e \cdot V \\ (\beta^-_{\mathcal{V}}) & \left\langle E \cdot V \middle\| \tilde{\lambda}\alpha.e \right\rangle \succ_{\beta^-_{\mathcal{V}}} & \langle V \| e\{E/\alpha\} \rangle \\ (\varsigma^-_{\mathcal{V}}) & e \cdot v \succ_{\varsigma^-_{\mathcal{V}}} & \mu\alpha. \langle v \| \tilde{\mu}y. \langle e \cdot y \| \alpha \rangle \rangle \qquad (v \notin Value_{\mathcal{V}}, x \text{ fresh}) \end{array}$$

and in call-by-name we have:

$$\begin{array}{lrl} & CoValue_{\mathcal{N}} ::= & \dots \mid \tilde{\lambda}\alpha.e \\ (\beta^-_{\mathcal{N}}) & \left\langle E \cdot V \middle\| \tilde{\lambda}\alpha.e \right\rangle \succ_{\beta^-_{\mathcal{N}}} & \langle V \| e\{E/\alpha\} \rangle \\ (\varsigma^-_{\mathcal{N}}) & e \cdot v \succ_{\varsigma^-_{\mathcal{N}}} & \mu\alpha. \langle \mu\beta. \langle \beta \cdot v \| \alpha \rangle \| e \rangle \qquad (e \notin CoValue_{\mathcal{N}}, \alpha \text{ fresh}) \end{array}$$

With the dual counterpart to functions in place, the full duality relationship of types and programs of the dual calculi is defined in Figure 3.21, where we assume an underlying involutive bijection $\overline{x}$ and $\overline{\alpha}$ between variables and co-variables.[9] First, notice that the duality operation is involutive on the nose: the dual of the dual is exactly the same as the original (Wadler, 2003).

Theorem 3.6 (Involutive duality)**.** *The duality operation* $\_^{\perp}$ *on environments, sequents, types, commands, terms, and co-terms is involutive, so that* $\_^{\perp\perp}$ *is the identity transformation.*

[9] By an involutive bijection, we mean that $\overline{x}$ gives a (co-)variable and $\overline{\alpha}$ gives a variable such that $\overline{x} \equiv \overline{y}$ and $\overline{\alpha} \equiv \overline{\beta}$ if and only if $x \equiv y$ and $\alpha \equiv \beta$, and also that $\overline{\overline{x}} \equiv x$ and $\overline{\overline{\alpha}} \equiv \alpha$.

Duality of sequents:

$$(c : (\Gamma \vdash \Delta))^{\perp} \triangleq c^{\perp} : (\Delta^{\perp} \vdash \Gamma^{\perp})$$

$$(\Gamma \vdash v : A \mid \Delta)^{\perp} \triangleq \Delta^{\perp} \mid v^{\perp} : A^{\perp} \vdash \Gamma^{\perp} \qquad (\Gamma \mid e : A \vdash \Delta)^{\perp} \triangleq \Delta^{\perp} \vdash e^{\perp} : A^{\perp} \mid \Gamma^{\perp}$$

$$(x_n : A_n, \dots, x_1 : A_1)^{\perp} \triangleq x_1^{\perp} : A_1^{\perp}, \dots, x_n^{\perp} : A_n^{\perp}$$

$$(\alpha_1 : A_1, \dots, \alpha_n : A_n)^{\perp} \triangleq \alpha_n^{\perp} : A_n^{\perp}, \dots, \alpha_1^{\perp} : A_1^{\perp}$$

Duality of types:

$$(X)^{\perp} \triangleq \overline{X}$$

$$(A \times B)^{\perp} \triangleq (A^{\perp}) + (B^{\perp}) \qquad (A + B)^{\perp} \triangleq (A^{\perp}) \times (B^{\perp})$$

$$(A \to B)^{\perp} \triangleq (B^{\perp}) - (A^{\perp}) \qquad (B - A)^{\perp} \triangleq (A^{\perp}) \to (B^{\perp})$$

$$(\forall X.A)^{\perp} \triangleq \exists X.(A^{\perp}) \qquad (\exists X.A)^{\perp} \triangleq \forall X.(A^{\perp})$$

$$(\neg A)^{\perp} \triangleq \neg(A^{\perp})$$

Duality of programs:

$$\langle v \| e \rangle^{\perp} \triangleq \left\langle e^{\perp} \middle\| v^{\perp} \right\rangle$$

$$(x)^{\perp} \triangleq \overline{x} \qquad [\alpha]^{\perp} \triangleq \overline{\alpha}$$

$$(\mu\alpha.c)^{\perp} \triangleq \tilde{\mu}\overline{\alpha}.c^{\perp} \qquad [\tilde{\mu}x.c]^{\perp} \triangleq \mu\overline{x}.c^{\perp}$$

$$(v_1, v_2)^{\perp} \triangleq \left[v_1^{\perp}, v_2^{\perp}\right] \qquad [e_1, e_2]^{\perp} \triangleq \left(e_1^{\perp}, e_2^{\perp}\right)$$

$$\iota_1\,(v)^{\perp} \triangleq \pi_1\left[v^{\perp}\right] \qquad \pi_1\,[e]^{\perp} \triangleq \iota_1\left(e^{\perp}\right)$$

$$\iota_2\,(v)^{\perp} \triangleq \pi_2\left[v^{\perp}\right] \qquad \pi_2\,[e]^{\perp} \triangleq \iota_2\left(e^{\perp}\right)$$

$$\mathsf{not}(e)^{\perp} \triangleq \mathsf{not}[e^{\perp}] \qquad \mathsf{not}[v]^{\perp} \triangleq \mathsf{not}(v^{\perp})$$

$$(\lambda x.v)^{\perp} \triangleq \tilde{\lambda}\overline{x}.[v^{\perp}] \qquad [\tilde{\lambda}\alpha.e]^{\perp} \triangleq \lambda\overline{\alpha}.(v^{\perp})$$

$$(e \cdot v)^{\perp} \triangleq e^{\perp} \cdot v^{\perp} \qquad [v \cdot e]^{\perp} \triangleq v^{\perp} \cdot e^{\perp}$$

$$(\Lambda X.v)^{\perp} \triangleq \tilde{\Lambda}X.[v^{\perp}] \qquad [\tilde{\Lambda}X.e]^{\perp} \triangleq \Lambda X.(e^{\perp})$$

$$(B \mathbin{@} v)^{\perp} \triangleq B^{\perp} \mathbin{@} [v^{\perp}] \qquad [B \mathbin{@} e]^{\perp} \triangleq B^{\perp} \mathbin{@} (e^{\perp})$$

FIGURE 3.21. The duality relation between the dual calculi.

Proof. By mutual induction on the definition of the duality operation $\_^{\perp}$ □

This relationship is not just a syntactic word game, but it gives us a duality between the typing derivations of terms and co-terms (Curien & Herbelin, 2000; Wadler, 2003):

Theorem 3.7 (Static duality)**.**

a) $c : (\Gamma \vdash \Delta)$ *is well-typed if and only if* $c^{\perp} : (\Delta^{\perp} \vdash \Gamma^{\perp})$ *is.*

b) $\Gamma \vdash v : A \mid \Delta$ *is well-typed if and only if* $\Delta^{\perp} \mid v^{\perp} : A^{\perp} \vdash \Gamma^{\perp}$ *is.*

c) $\Gamma \mid e : A \vdash \Delta$ *is well-typed if and only if* $\Delta^{\perp} \vdash e^{\perp} : A^{\perp} \mid \Gamma^{\perp}$ *is.*

Furthermore, if a command, term, or co-term lies in the LKQ sub-syntax, its dual lies in the LKT sub-syntax and vice versa.

Proof. By induction on the typing derivation. □

The dynamic aspect of duality takes form as a relationship between the two reduction systems for evaluating programs: call-by-value reduction is dual to call-by-name reduction. That is, if we have a command c that behaves a certain way according to the call-by-value calculus, then the dual command $c^{\perp}$ behaves in a correspondingly dual way according to the call-by-name calculus, and vice versa. The two dynamic semantics (operational, reduction, and equational) mirror each other exactly, rule for rule (Curien & Herbelin, 2000; Wadler, 2003).

Theorem 3.8 (Dynamic duality)**.** *a)* $c \succ_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}} c'$ *if and only if* $c^{\perp} \succ_{\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}\beta_{\mathcal{N}}} c'^{\perp}$*, and dually* $c \succ_{\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}\beta_{\mathcal{N}}} c'$ *if and only if* $c^{\perp} \succ_{\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}\beta_{\mathcal{V}}} c'^{\perp}$*.*

b) $v \succ_{\eta_{\mu}\varsigma_{\mathcal{V}}} v'$ *if and only if* $v^{\perp} \succ_{\eta_{\tilde{\mu}}\varsigma_{\mathcal{N}}} v'^{\perp}$*, and dually* $v \succ_{\eta_{\mu}\varsigma_{\mathcal{N}}} v'$ *if and only if* $v^{\perp} \succ_{\eta_{\tilde{\mu}}\varsigma_{\mathcal{V}}} v'^{\perp}$*.*

c) $e \succ_{\eta_{\tilde{\mu}}\varsigma_{\mathcal{V}}} e'$ *if and only if* $e^{\perp} \succ_{\eta_{\mu}\varsigma_{\mathcal{N}}} e'^{\perp}$*, and dually* $e \succ_{\eta_{\tilde{\mu}}\varsigma_{\mathcal{N}}} e'$ *if and only if* $e^{\perp} \succ_{\eta_{\mu}\varsigma_{\mathcal{V}}} e'^{\perp}$*.*

Proof. By cases on the respective rewriting rules, using the fact that substitution commutes with duality ($(c\,\{V/x\})^{\perp} =_{\alpha} c^{\perp}\left\{V^{\perp}/\overline{x}\right\}$, $(c\,\{E/\alpha\})^{\perp} =_{\alpha} c^{\perp}\left\{E^{\perp}/\overline{\alpha}\right\}$, $(c\,\{A/X\})^{\perp} =_{\alpha} c^{\perp}\left\{A^{\perp}/X\right\}$, and similarly for (co-)terms) which is guaranteed by the fact that the duality operation is compositional and hygienic (Downen & Ariola, 2014a). □

CHAPTER IV

Polarity

Looking back to Gentzen's original LK from Figure 3.5, a careful eye might notice that there is a bit of an inconsistency among the logical rules. In particular, compare left implication introduction ($\supset L$) with right conjunction ($\wedge R$) and left disjunction ($\vee L$) introduction and notice how they treat their auxiliary propositions (hypotheses Γ and consequences Δ) very differently. In both the $\wedge R$ and $\vee L$ rules, the auxiliary propositions are shared among both premises and the deduction: each sequent contains exactly the same extra hypotheses (Γ) and consequences (Δ). However, the $\supset L$ rule does not follow this pattern. Instead, the two premises of the $\supset L$ rule contain *different* auxiliary propositions from one another, which are then combined together in the deduction: each sequent contains potentially different hypotheses and consequences.

Why are the rules for implication appear so different from the rules for conjunction and disjunction? Is this merely a notational accident, or is there some significance to the way these side propositions are threaded through the proof tree? As it turns out, we can classify the logical connectives in a way that emphasizes this distinction, which through the Curry-Howard lense has a profound impact on our understanding of the computational nature of the sequent calculus. Before in Chapter III, we found that the sequent calculus shows us the duality between evaluation strategies—namely the call-by-value and call-by-name strategies—via two distinct languages with the same syntax but different semantics. This distinction between the semantics of the dual call-by-value and call-by-name calculi becomes apparent when we consider the operational behavior of programs. For example, Wadler (2003) was able to encode functions in terms of the other connectives, but surprisingly *different* encodings are necessary for both call-by-value and call-by-name. Even though they share a syntax, the two dual calculi truly describe different languages. Instead, we will soon find that an alternative interpretation of the sequent calculus lets us express the same duality of evaluation *within* the same language, so that a single program might employ *both* call-by-value and call-by-name during its execution.

Additive and Multiplicative LK

Recall back to the basic left introduction inference rules for conjunction in Figure 3.2. These rules state that if A is false then $A \wedge B$ is false, and likewise if B is false then $A \wedge B$ is false as well. However, there is another presentation of conjunction that makes use of the internal structure of sequents. We originally decided in Chapter III to interpret a sequent as meaning that the truth of *all* hypotheses entails the truth of *one* consequence. So for example, the sequent $A, B \vdash C, D$ means that "A and B entails C or D." In other words, the commas to the left are pronounced as "and," and the commas to the right are pronounced "or."

We might then formalize this interpretation by saying that the logical connective for conjunction actually corresponds to a comma on the left, so that the sequents $A, B \vdash$ and $A \wedge B \vdash$ are equally valid as shown by the two inferences which reverse one another (from bottom-up to top-down):

$$\frac{A, B \vdash}{A \wedge B \vdash} \qquad\qquad \frac{A \wedge B \vdash}{A, B \vdash}$$

Notice how the sequents $A, B \vdash$ and $A \wedge B \vdash$ are equivalent statements since both mean that "A and B entails false," which justifies that the above inference rules are valid. Likewise, we could equate the logical connective for disjunction with a comma on the right, so that the sequents $\vdash A, B$ and $\vdash A \vee B$ are equally valid as shown by the inferences:

$$\frac{\vdash A, B}{\vdash A \vee B} \qquad\qquad \frac{\vdash A \vee B}{\vdash A, B}$$

This gives an alternative to the right introduction rules for disjunction in contrast to the ones given in Figure 3.3.

Notice how the above alternative rules for conjunction and disjunction are reversible: both the top-down and bottom-up inferences are valid. More generally, an inference of the form

$$\frac{H_1 \quad H_2 \quad \ldots \quad H_n}{J}$$

is *reversible* when there are derivations $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n$ for each of

$$\begin{array}{c} J \\ \vdots\ \mathcal{D}_1 \\ H_1 \end{array} \qquad \begin{array}{c} J \\ \vdots\ \mathcal{D}_2 \\ H_2 \end{array} \qquad \ldots \qquad \begin{array}{c} J \\ \vdots\ \mathcal{D}_3 \\ H_n \end{array}$$

and *irreversible* otherwise. So we can say that the above alternative left introduction of conjunction and right introduction of disjunction are both reversible. These formulations of conjunction and disjunction contrast with the rules that were given in Figures 3.2 and 3.3. Clearly $A \vdash$ (i.e. "A is false") is a much stronger statement than $A \wedge B \vdash$ (i.e. "the conjunction of A and B is false"), so the left introduction rules given in Figure 3.2 are irreversible. Likewise, $\vdash A$ (i.e. "A is true") is a much stronger statement than $\vdash A \vee B$ (i.e. "either A or B is true"), so the right introduction rules for conjunction given in Figure 3.3 are also irreversible.

It seems that we have a substantive choice on how we might phrase conjunction and disjunction in the setting of the sequent calculus. Instead of just arbitrarily choosing one of them, we can consider all the possibilities at once in the same logic as shown in Figure 4.1. In this combined logic, we have two separate logical connectives for conjunction and two connectives for disjunction. Additionally, there are two separate constants (i.e. nullary connectives) for truth and falsehood. Our original formulation of conjunction ($\wedge$) and disjunction ($\vee$) in LK from Figure 3.5 are preserved as the $\&$ and $\oplus$ connectives, respectively, as well as truth ($\top$) and falsehood ($\bot$) which go by the same name. The new alternatives for truth, falsehood, conjunction, and disjunction discussed above are denoted by the 1, 0, $\otimes$, and $\parr$ connectives, respectively. Finally, the presentation of negation ($\neg$) and implication ($\supset$) is unchanged. For now we delay further discussion of the quantifiers until Chapter VI.

Now we can more formally analyze the reversibility of the logical rules for the different variations of the connectives. The left introduction for $\otimes$-conjunction and $\oplus$-disjunction are reversible because the sequent $\Gamma, A, B \vdash \Delta$ follows from $\Gamma, A \otimes B \vdash \Delta$ and each of $\Gamma, A \vdash \Delta$ and $\Gamma, B \vdash \Delta$ follow from $\Gamma, A \oplus B \vdash \Delta$:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,Ax \quad \dfrac{}{B \vdash B}\,Ax}{A, B \vdash A \otimes B}\,\otimes R \qquad \Gamma, A \otimes B \vdash \Delta}{\Gamma, A, B \vdash \Delta}\,Cut$$

$$A, B, C \in \mathit{Proposition} ::= X \mid 0 \mid 1 \mid A \oplus B \mid A \otimes B \mid \top \mid \bot \mid A \mathbin{\&} B \mid A \parr B \mid A \supset B \mid \neg A$$

$$\Gamma \in \mathit{Hypothesis} ::= A_1, \ldots, A_n \qquad \Delta \in \mathit{Consequence} ::= A_1, \ldots, A_n$$

$$\mathit{Judgement} ::= \Gamma \vdash \Delta$$

Axiom and cut:

$$\frac{}{A \vdash A}\,\mathit{Ax} \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma', \Gamma' \vdash \Delta', \Delta}\,\mathit{Cut}$$

Logical rules:

$$\frac{}{\vdash 1}\,1R \qquad \frac{\Gamma \vdash \Delta}{\Gamma, 1 \vdash \Delta}\,1L \qquad \text{no } 0R \text{ rule} \qquad \frac{}{\Gamma, 0 \vdash \Delta}\,0L$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash A \otimes B, \Delta, \Delta'}\,{\otimes}R \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta}\,{\otimes}L$$

$$\frac{}{\Gamma \vdash \top, \Delta}\,{\top}R \qquad \text{not } {\top}L \text{ rule} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta}\,{\bot}R \qquad \frac{}{\bot \vdash}\,{\bot}L$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \mathbin{\&} B, \Delta}\,\&R \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A \mathbin{\&} B \vdash \Delta}\,\&L_1 \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \mathbin{\&} B \vdash \Delta}\,\&L_2$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta}\,{\oplus}R_1 \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta}\,{\oplus}R_2 \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta}\,{\oplus}L$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \parr B, \Delta}\,{\parr}R \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta'}{\Gamma, \Gamma', A \parr B \vdash \Delta, \Delta'}\,{\parr}L$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,{\neg}R \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,{\neg}L \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta}\,{\supset}R \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \supset B \vdash \Delta, \Delta'}\,{\supset}L$$

Structural rules:

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\,\mathit{WR} \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\,\mathit{WL} \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\,\mathit{CR} \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\,\mathit{CL}$$

$$\frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, B, A, \Delta'}\,\mathit{XR} \qquad \frac{\Gamma, A, B, \Gamma' \vdash \Delta}{\Gamma, B, A, \Gamma' \vdash \Delta}\,\mathit{XL}$$

FIGURE 4.1. An additive and multiplicative LK sequent calculus: with two truths (1, $\top$), two falsehoods (0, $\bot$), two conjunctions ($\otimes$, &), two disjunctions ($\oplus$, $\parr$), one negation ($\neg$), and one implication ($\supset$).

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathit{Ax}}{A \vdash A \oplus B}\,{\oplus}R_1 \qquad \Gamma, A \oplus B \vdash \Delta}{\Gamma, A \vdash \Delta}\,\mathit{Cut} \qquad\qquad \dfrac{\dfrac{\dfrac{}{B \vdash B}\,\mathit{Ax}}{B \vdash A \oplus B}\,{\oplus}R_2 \qquad \Gamma, A \oplus B \vdash \Delta}{\Gamma, B \vdash \Delta}\,\mathit{Cut}$$

However, the right rules of ⊗-conjunction and ⊕-disjunction are irreversible because the premises are stronger than the conclusion. Clearly neither $\Gamma \vdash A, \Delta$ nor $\Gamma \vdash B, \Delta$ follow from the weaker sequent $\Gamma \vdash A \oplus B, \Delta$, but also neither $\Gamma \vdash A, \Delta$ nor $\Gamma' \vdash B, \Delta'$ follow from $\Gamma, \Gamma' \vdash A \otimes B, \Delta, \Delta'$ because of the way that the side-propositions Γ, Γ' and Δ, Δ' from the conclusion are split up between the two premises.

In contrast, the right introduction rules for &-conjunction, ⅋-disjunction, and →-implication are reversible because the premises are weak enough to be proved from the conclusions:

$$\dfrac{\Gamma \vdash A \mathbin{\&} B, \Delta \qquad \dfrac{\dfrac{}{A \vdash A}\,\mathit{Ax}}{A \mathbin{\&} B \vdash A}\,{\&}L_1}{\Gamma \vdash A, \Delta}\,\mathit{Cut} \qquad\qquad \dfrac{\Gamma \vdash A \mathbin{\&} B, \Delta \qquad \dfrac{\dfrac{}{B \vdash B}\,\mathit{Ax}}{A \mathbin{\&} B \vdash B}\,{\&}L_2}{\Gamma \vdash B, \Delta}\,\mathit{Cut}$$

$$\dfrac{\Gamma \vdash A \parr B, \Delta \qquad \dfrac{\dfrac{}{A \vdash A}\,\mathit{Ax} \quad \dfrac{}{B \vdash B}\,\mathit{Ax}}{A \parr B \vdash A, B}\,{\parr}L}{\Gamma \vdash A, B, \Delta}\,\mathit{Cut} \qquad\qquad \begin{array}{c} \dfrac{\Gamma \vdash A \supset B, \Delta \qquad \dfrac{\dfrac{}{A \vdash A}\,\mathit{Ax} \quad \dfrac{}{B \vdash B}\,\mathit{Ax}}{A, A \supset B \vdash B}\,{\supset}L}{\Gamma, A \vdash \Delta, B}\,\mathit{Cut} \\ \vdots\ \mathit{XR} \\ \Gamma, A \vdash B, \Delta \end{array}$$

However, each of the &-conjunction, ⅋-disjunction, and ⊃-implication left introduction rules are irreversible for similar reasons as the right introduction rules for ⊗-conjunction and ⊕-disjunction. Clearly neither $\Gamma, A \vdash \Delta$ nor $\Gamma, B \vdash \Delta$ follow from the weaker sequent $\Gamma, A \mathbin{\&} B \vdash \Delta$. Furthermore, both the ${\parr}L$ and ${\supset}L$ share the same splitting problem that causes the irreversibility of ${\otimes}R$.

One consequence of reversibility is that any derivation whose conclusion matches the conclusion of a reversible rule *might as well* end with that reversible rule, because we can always extract out the premises to the rule and then reassemble the same conclusion. For example, suppose that we have a derivation $\mathcal{D}$ of the sequent $\Gamma \vdash A \mathbin{\&} B, \Delta$, where the proposition $A \mathbin{\&} B$ appears on the right side. Then by the reversibility of the ${\&}R$ rule noted above, we have derivations from $\Gamma \vdash A \mathbin{\&} B, \Delta$ to $\Gamma \vdash A, \Delta$ and $\Gamma \vdash A, \Delta$, which we will denote by the names ${\&}{R_1}^{-1}$ and ${\&}{R_2}^{-1}$ respectively. These two reverse derivations let us expand $\mathcal{D}$ to get an extended derivation which ends with

$\&R$ as follows:

$$\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma \vdash A \mathbin{\&} B, \Delta\end{array} \;\prec\; \dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma \vdash A \mathbin{\&} B, \Delta\\ \vdots\,{\&R_1}^{-1}\\ \Gamma \vdash A, \Delta\end{array} \qquad \begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma \vdash A \mathbin{\&} B, \Delta\\ \vdots\,{\&R_2}^{-1}\\ \Gamma \vdash B, \Delta\end{array}}{\Gamma \vdash A \mathbin{\&} B, \Delta}\,\&R$$

Similarly, we can expand arbitrary derivations of sequents with $A \parr B$ or $A \to B$ on the right side using the derivations $\parr R^{-1}$ and $\to R^{-1}$ which reverse the $\parr R$ and $\to R$ right introduction rules:

$$\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma \vdash A \parr B, \Delta\end{array} \;\prec\; \dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma \vdash A \parr B, \Delta\\ \vdots\,\parr R^{-1}\\ \Gamma \vdash A, B, \Delta\end{array}}{\Gamma \vdash A \parr B, \Delta}\,\parr R \qquad \begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma \vdash A \to B, \Delta\end{array} \;\prec\; \dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma \vdash A \to B, \Delta\\ \vdots\,\to R^{-1}\\ \Gamma, A \vdash B, \Delta\end{array}}{\Gamma \vdash A \to B, \Delta}\,\to R$$

The same expansion also occurs when the proposition $A \oplus B$ or $A \otimes B$ appears on the left of the concluding sequent, by using the ${\oplus L_1}^{-1}$, ${\oplus L_2}^{-1}$, and $\otimes L^{-1}$ reverse derivations of the $\oplus L$ and $\otimes L$ left introduction rules.

$$\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma, A \oplus B \vdash \Delta\end{array} \;\prec\; \dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma, A \oplus B \vdash \Delta\\ \vdots\,{\oplus L_1}^{-1}\\ \Gamma, A \vdash \Delta\end{array} \qquad \begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma, A \oplus B \vdash \Delta\\ \vdots\,{\oplus L_2}^{-1}\\ \Gamma, B \vdash \Delta\end{array}}{\Gamma, A \oplus B \vdash \Delta}\,\oplus L$$

$$\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma, A \otimes B \vdash \Delta\end{array} \;\prec\; \dfrac{\begin{array}{c}\vdots\,\mathcal{D}\\ \Gamma, A \otimes B \vdash \Delta\\ \vdots\,\otimes L^{-1}\\ \Gamma, A, B \vdash \Delta\end{array}}{\Gamma, A \otimes B \vdash \Delta}\,\otimes L$$

So in comparison with natural deduction, whereas the steps of cut elimination (Section 3.1) in the sequent calculus correspond with local soundness (Section 2.1), the above reversibility expansions correspond with local completeness.

With both variations of the connectives included in a single logic, we can compare and contrast them by the emergent properties of their logical rules. Notice how the auxiliary hypotheses Γ and consequences Δ in the $\&R$ and $\oplus L$ rules are shared among both premises as well in the conclusion, so that Γ and Δ are "copied" when the rules are read from the bottom-up. Because the side-propositions are copied bottom-up, we say that the &-conjunction and $\oplus$-disjunction are *additive* connectives. In contrast, in each of the $\otimes R$, $⅋ L$, and $\supset L$ rules the two premises contain different auxiliary hypotheses and consequences which are "merged" when the rules are read from the top-down. Because the side-propositions are merged top-down, we say that the $\otimes$-conjunction, $⅋$-disjunction, and $\supset$-implication are *multiplicative* connectives. In the degenerate case for the nullary connectives, we can say that $\top$ and 0 are additive because the Γ and Δ in the conclusion of their only introduction rule ($\top R$ and $\bot L$) is "copied" among its zero premises, whereas the 1 and 0 have rules ($1R$ and $0L$) that "merge" the hypothesis and conclusions from their zero premises into the conclusion. Note that $\neg$-negation is neither additive nor multiplicative—or perhaps it could be considered *both* additive and multiplicative—since both its right and left introduction rules have exactly one premise.

Besides the additive-multiplicative distinction, there is another axis which is perhaps more fundamental upon which we can classify the connectives. Recall the previous discussion of *reversibility* of the inference rules that lead us to consider $\otimes$-conjunction and $⅋$-disjunction as alternatives to the &-conjunction and $\oplus$-conjunction that were inherited from Gentzen's LK. Both the $\otimes$-conjunction and $\oplus$-disjunction have reversible left introductions because the premises are weak enough to be proved from the conclusion. On the flip side, we saw that &-conjunction, $⅋$-disjunction, and $\supset$-implication have reversible right introductions for dual reasons. We can thus divide the logical connectives based on two *polarities*: connectives with reversible left introductions and irreversible right introductions are *positive*, and dually connectives with reversible right introductions and irreversible left introductions are *negative*. Based on our previous analysis, we can say that $\otimes$-conjunction and $\oplus$-disjunction are positive, whereas &-conjunction, $⅋$-disjunction, and $\supset$-disjunction are negative. Note again that $\neg$-negation does not directly participate in this classification and is neutral with regard to polarity because both the left and right $\neg$ introductions are reversible, making it both—or neither, depending on our perspective—positive and negative at the same time. We can thus categorize all the binary connectives along the additive-multiplicative and positive-negative axes, as shown in Figure 4.2. These two classifications are enough to separate all of the connectives into different quadrants

	Positive	Negative
Additive	$\oplus$	$\&$
Multiplicative	$\otimes$	$\parr, \supset$

FIGURE 4.2. The positive/negative and additive/multiplicative classification of binary connectives.

based on their properties, so that only $\parr$ and $\supset$ share the same quadrant showing that these are the two connectives that are most similar to one another.

Pattern Matching and Extensionality

Let us now consider a language for the additive and multiplicative LK sequent calculus which is well suited for expressing the polarity of connectives within the form of its expressions. The language shown in Figure 4.3, which extends the core $\mu\tilde{\mu}$-calculus from Figure 3.7, is based on Munch-Maccagnoni's (2009) system L family of calculi.[1] System L is visually rather different from the dual calculi we studied previously in Chapter III, where its most obvious first departure from the dual calculi is its pervasive use of *pattern-matching* as a core language construct.

One way to understand the role of pattern-matching in programming and its connection to polarity is to look at Dummett's 1976 lectures (Dummett, 1991) on the justification of logical principles. In essence, Dummett suggested that there are effectively two ways for framing the meaning of logical laws, which reveals a certain bias in the logician: the *verificationist* and the *pragmatist*.

In the eyes of a verificationist, it is the rules for proving a proposition (corresponding to the right introduction rules in either natural deduction or sequent calculus) that give meaning to a logical connective. These are the primitive rules for a connective that define its character. All the other rules of a connective (the elimination or left rules) must then be justified with respect to its right introductions. In other words, the meaning of a proposition can be devised from its *canonical proofs* (Prawitz, 1974) composed of right introduction rules, and the other rules are sound with respect to them. This is an alternative to the global property of cut elimination from Section 3.1 that is more similar to local soundness for natural deduction described in Section 2.1.

[1] We consider here the two-sided variant of system L to make easier comparisons with the other languages for the sequent calculus.

$$
\begin{aligned}
A, B, C \in \mathit{Type} &::= X \mid 0 \mid 1 \mid A \oplus B \mid A \otimes B \mid {\sim}A \\
&\mid \top \mid \bot \mid A \mathbin{\&} B \mid A \parr B \mid A \to B \mid \neg A \\
v \in \mathit{Term} &::= x \mid \mu\alpha.c \mid () \mid \iota_1\,(v) \mid \iota_2\,(v) \mid (v, v) \mid {\sim}\,(e) \\
&\mid \mu([].c) \mid \mu(\pi_1\,[\alpha].c \mid \pi_2\,[\beta].c) \mid \mu([\alpha, \beta].c) \mid \mu([x \cdot \beta].c) \mid \mu(\neg\,[x].c) \\
e \in \mathit{CoTerm} &::= \alpha \mid \tilde{\mu}x.c \mid \tilde{\mu}[().c] \mid \tilde{\mu}[\iota_1\,(x).c \mid \iota_2\,(y).c] \mid \tilde{\mu}[(x, y).c] \mid \tilde{\mu}[{\sim}\,(\alpha).c] \\
&\mid \pi_1\,[e] \mid \pi_2\,[e] \mid [e, e] \mid v \cdot e \mid \neg\,[v] \\
c \in \mathit{Command} &::= \langle v \| e \rangle
\end{aligned}
$$

Logical rules:

$$
\frac{}{\vdash () : 1 \mid}\ 1R
\qquad
\frac{c : (\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}[().c] : 1 \vdash \Delta}\ 1L
\qquad
\text{not } 0R \text{ rule}
\qquad
\frac{}{\Gamma \mid \tilde{\mu}[] : 0 \vdash \Delta}
$$

$$
\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_2\,(v) : A \oplus B \mid \Delta}\ \oplus R_1
\qquad
\frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \iota_1\,(v) : A \oplus B \mid \Delta}\ \oplus R_2
$$

$$
\frac{c : (\Gamma, x : A \vdash \Delta) \quad c' : (\Gamma, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}[\iota_1\,(x).c \mid \iota_2\,(y).c'] : A \oplus B \vdash \Delta}\ \oplus L
$$

$$
\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash v' : B \mid \Delta}{\Gamma, \Gamma' \vdash (v, v') : A \otimes B \mid \Delta, \Delta'}\ \otimes R
\qquad
\frac{c : (\Gamma, x : A, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}[(x, y).c] : A \otimes B \vdash \Delta}\ \otimes L
$$

$$
\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash {\sim}\,(e) : {\sim}A \mid \Delta}\ {\sim}R
\qquad
\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \mid \mu({\sim}\,(\alpha).c) : {\sim}A \vdash \Delta}\ {\sim}L
$$

$$
\frac{}{\Gamma \vdash \mu() : \top \mid \Delta}\ \top R
\qquad
\text{no } \top L \text{ rule}
\qquad
\frac{c : (\Gamma \vdash \Delta)}{\Gamma \vdash \mu([].c) : \bot \mid \Delta}\ \bot R
\qquad
\frac{}{\mid [] : \bot \vdash}\ \bot L
$$

$$
\frac{c : (\Gamma \vdash \alpha : A, \Delta) \quad c' : (\Gamma \vdash \beta : B, \Delta)}{\Gamma \vdash \mu(\pi_1\,[\alpha].c \mid \pi_2\,[\beta].c') : A \mathbin{\&} B \mid \Delta}\ \& R
$$

$$
\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \pi_1\,[e] : A \mathbin{\&} B \vdash \Delta}\ \& L_1
\qquad
\frac{\Gamma \mid e : B \vdash \Delta}{\Gamma \mid \pi_2\,[e] : A \mathbin{\&} B \vdash \Delta}\ \& L_2
$$

$$
\frac{c : (\Gamma \vdash \alpha : A, \beta : B, \Delta)}{\Gamma \vdash \mu([\alpha, \beta].c) : A \parr B \mid \Delta}\ \parr R
\qquad
\frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid e' : B \vdash \Delta}{\Gamma, \Gamma' \mid [e, e'] : A \parr B \vdash \Delta, \Delta'}\ \parr L
$$

$$
\frac{c : (\Gamma, x : A \vdash \beta : B, \Delta)}{\Gamma \vdash \mu([x \cdot \beta].c) : A \to B \mid \Delta}\ {\to}R
\qquad
\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : B \vdash \Delta'}{\Gamma, \Gamma' \mid v \cdot e : A \to B \vdash \Delta, \Delta'}\ {\to}L
$$

$$
\frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \vdash \mu(\neg\,[x].c) : \neg A \mid \Delta}\ \neg R
\qquad
\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \mid \neg\,[v] : \neg A \vdash \Delta}\ \neg L
$$

FIGURE 4.3. The syntax and types semantics for system L: with two unit types (1, ⊤), two empty types (0, ⊥), (co-)products (&, ⊕), (co-)pairs (⊗, ⅋), two negations (∼, ¬), and functions (→).

In the eyes of a pragmatist, it is the rules for using a proposition (corresponding to the elimination rules in natural deduction left rules in sequent calculus), that give meaning to a logical connective. That is to say, the primitive concept is what can be done with a proposition. This stance is the polar opposite of the verificationist. For a pragmatist, canonical proofs are composed of elimination or left rules , and the other rules must be sound with respect to the way assumptions are used rather than the way facts are verified. The key insight behind this connection is that the positive connectives follow a verificationist's point of view, whereas the negative connectives follow a pragmatist's point of view. In terms of system L, positive types focus on the patterns or shapes of terms (which create results) whereas negative types focus on the patterns or shapes of co-terms (which use results).

Since the positive connectives correspond to a verificationist style of proof, the proofs (i.e. verifications) of a proposition fall within a fixed set of well-known canonical forms, whereas the uses (i.e. refutations) of a proposition are arbitrary. Therefore, in a program corresponding to a verificationist proof, the terms for producing output also must fall within a fixed set of forms, but the co-terms for consuming input are allowed to be arbitrary. In order to gain a foot-hold on the unrestricted nature of positive co-terms, we may describe them by *inversion* on the possible forms of their input. That is to say, positive co-terms may be defined by *cases* on the structure of all possible input they might receive. In other words, positive types follow the general pattern that terms are formed by construction, whereas co-terms are formed by case analysis on term constructors

Compared to the positive connectives, the pragmatist approach to negative connectives may seem a bit unusual. Rather than thinking about how to conclude true facts, the pragmatist takes the dual approach and focuses attention on how to make use of those facts. In this way, the methods of using an assumed proposition are limited to a fixed set of known canonical forms, whereas the conclusions of a proposition may be arbitrary. The programs that correspond with pragmatist proofs are likewise dual to verificationist proofs, so that the relative roles of producers and consumers are reversed. In a pragmatist program, the terms that produce output are allowed to have an arbitrary form. Instead, it is the co-terms for consuming input that must fall within a fixed set of known forms—the legal observations of a type. We may then define terms by inversion on the possible forms of their consumer, so that they are given by cases on the observation of their output. In other words, general pattern for negative connectives is that the co-terms are formed by construction, whereas the terms are formed by *the dual form of case analysis* on co-term constructors.

For example, in the dual calculi, a value of the product type $A \times B$ was created by the pair term (v_1, v_2) which are used by a projection co-term of the form $\pi_1\,[e]$ or $\pi_2\,[e]$. In system L, however, we have two different methods to conjoin two types. From the verificationist viewpoint, the positive $A \otimes B$ method to conjunction puts the focus on the construction of pairs representing the canonical proof of a conjunction of two parts, keeping terms of the form $(v_1, v_2) : A \otimes B$ that clearly contains both $v_1 : A$ and $v_1 : B$ sub-terms, as the single right introduction rule defining $A \otimes B$:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash v' : B \mid \Delta}{\Gamma, \Gamma' \vdash (v, v') : A \otimes B \mid \Delta, \Delta'} \otimes R$$

To use a value of type $A \otimes B$, a co-term only needs to justify its reaction to the canonical pair values, such as the *case abstraction* co-term $\tilde{\mu}[(x, y).c] : A \otimes B$ that performs a pattern-matching case analysis to bind $x : A$ to the first component and $y : B$ to the second component of its given pair in the arbitrary command c:

$$\frac{c : (\Gamma, x : A, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}[(x, y).c] : A \otimes B \vdash \Delta} \otimes L$$

From the pragmatist viewpoint, the negative $A \,\&\, B$ method to conjunction puts the focus on the destruction of pairs, keeping co-terms of the form $\pi_1\,[e] : A \,\&\, B$ and $\pi_2\,[e] : A \,\&\, B$ that clearly mark the choice between the two canonical left introduction rules of products defining $A \,\&\, B$:

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \pi_1\,[e] : A \,\&\, B \vdash \Delta} \,\&L_1 \qquad\qquad \frac{\Gamma \mid e : B \vdash \Delta}{\Gamma \mid \pi_2\,[e] : A \,\&\, B \vdash \Delta} \,\&L_2$$

To create a value of type $A\&B$, a term only needs to justify its reaction the two possible projection observations, such as the *co-case abstraction* term $\mu(\pi_1\,[\alpha].c_1 \mid \pi_2\,[\beta].c_2) : A \,\&\, B$ that performs pattern-matching case analysis which projection is observing it, binding $\alpha : A$ to $e_1 : A$ in c_1 in the case of a $\pi_1\,[e_1]$ projection and binding $\beta : B$ to $e_2 : B$ in c_2 in the case of a $\pi_2\,[e_2]$ projection:

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta) \quad c' : (\Gamma \vdash \beta : B, \Delta)}{\Gamma \vdash \mu(\pi_1\,[\alpha].c \mid \pi_2\,[\beta].c') : A \,\&\, B \mid \Delta} \,\&R$$

As another example, the dual calculi creates values of the sum type $A + B$ by the injection terms $\iota_1\,(v)$ and $\iota_2\,(v)$ which are used by a co-pair co-term of the form $[v_1, v_2]$, and in system L each of these two constructions show up separately in the two different methods to disjoin two types. On the one hand, the $A \oplus B$ method to disjunction keeps the injection terms $\iota_1\,(v) : A \oplus B$ and $\iota_2\,(v) : A \oplus B$ that clearly mark the choice between the canonical right introduction rules defining $A \oplus B$:

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_2\,(v) : A \oplus B \mid \Delta}\ \oplus R_1 \qquad\qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \iota_1\,(v) : A \oplus B \mid \Delta}\ \oplus R_2$$

To use a value of type $A \oplus B$, we only need to justify the reaction of a co-term to the canonical injection terms, such as the case abstraction co-term $\tilde{\mu}[\iota_1\,(x).c_1 \mid \iota_2\,(y).c_2] : A \oplus B$ that checks which injection it receives, binding $x : A$ to $v_1 : A$ in c_1 in the case of $\iota_1\,(v_1)$ and binding $y : B$ to $v_2 : B$ in c_2 in the case of $\iota_2\,(v_2)$:

$$\frac{c : (\Gamma, x : A \vdash \Delta) \quad c' : (\Gamma, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}[\iota_1\,(x).c \mid \iota_2\,(y).c'] : A \oplus B \vdash \Delta}\ \oplus L$$

On the other hand, the $A \parr B$ method of disjunction puts the focus on the destruction of sums, keeping co-terms of the form $[e_1, e_2]$ that clearly contains both $e_1 : A$ and $e_2 : B$ sub-co-terms, as the single canonical left introduction rule defining $A \parr B$:

$$\frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid e' : B \vdash \Delta}{\Gamma, \Gamma' \mid [e, e'] : A \parr B \vdash \Delta, \Delta'}\ \parr L$$

To create a value of type $A \parr B$, we only need to justify the reaction of a term to the canonical co-pair observations, such as the co-case abstraction term $\mu([\alpha, \beta].c) : A \parr B$ that binds $\alpha : A$ to first component and $\beta : B$ to the second component of its given co-pair in the arbitrary command c:

$$\frac{c : (\Gamma \vdash \alpha : A, \beta : B, \Delta)}{\Gamma \vdash \mu([\alpha, \beta].c) : A \parr B \mid \Delta}\ \parr R$$

The rest of the connectives follow suit accordingly, where positive connectives construct terms according to certain patterns and have co-terms which match on those patterns by case analysis, and negative connectives construct co-terms according to certain patterns and have terms which match on those patterns by case analysis. The positive constants 1 and 0 are nullary versions of $A \otimes B$ and $A \oplus B$ so they contain the nullary versions of pairs and co-products. The negative constants $\top$ and $\bot$ are nullary versions of $A \mathbin{\&} B$ and $A \parr B$ so they contain the nullary versions of products and co-pairs. Functions $A \to B$ are another example of a multiplicative negative type like the negative disjunction $A \parr B$, and so it contains similar (co-)terms for the sake of uniformity. This means that the call stacks $v \cdot e : A \to B$ for functions are the same as in the dual calculi, but λ-abstractions have been replaced with the co-case abstraction terms $\mu([x \cdot \beta].c) : A \to B$ which deconstruct a call stack to bind the argument to $x : A$ and the return co-term to $\beta : B$ in the command c. Note that this change in representation from λ-abstractions to call stack deconstructions does not change the expressiveness of functions, since the each can represent the other as macro expansions:

$$\mu([x \cdot \beta].c) = \lambda x.\mu\beta.c \qquad \lambda x.v = \mu([x \cdot \beta].\langle v \| \beta \rangle) \qquad (\beta \notin FV(v))$$

Finally, we have to accomodate negation, which could be considered both positive and negative as we previous saw in Section 4.1. Therefore, instead of breaking the pattern or choosing arbitrarily, we include two different negation connectives—a positive negation $\sim\!A$ and a negative negation $\neg A$—to express the two possible orientations of construction and deconstruction by case analysis.

Remark 4.1. It is worthwhile to pause and ask why the pragmatist representation of logical connectives may appear to be backwards. For example, $\parr$ is a logical "or" whose interpretation appears to be an "and" combination of two things, whereas & is a logical "and" whose interpretation appears to be an "or" choice of two alternatives. The reason is that the pragmatist approach requires us to completely reverse the way we think about proving. Under the verificationist approach, we focus on how to establish truth: to show that "A and B" is true, we need to show that *both* A and B are true; to show that "A or B" is true, it suffices to show that *either* A is true or B is true. Instead, the pragmatist approach asks us to focus on the ways to establish falsehood: to show that "A and B" is false, it suffices to show that *either* A is false or B is false; to show that "A or B" is false we need to show that *both* A and B are false. Whereas the verificationist is primarily concerned with building a *proof*, the pragmatist is instead concerned with building a *refutation*. Therefore, the pragmatist interpretation of negative connectives intuitively has a negative baked in: "and" is

Positive $\eta_{\mathcal{P}}$ rules:

$$\left.\begin{array}{lr@{\;}l}
(\eta_{\mathcal{P}}^{0}) & e : 0 \prec_{\eta_{\mathcal{P}}^{0}} & \tilde{\mu}[] \\
(\eta_{\mathcal{P}}^{1}) & e : 1 \prec_{\eta_{\mathcal{P}}^{1}} & \tilde{\mu}[().\langle ()\|e\rangle] \\
(\eta_{\mathcal{P}}^{\oplus}) & e : A \oplus B \prec_{\eta_{\mathcal{P}}^{\oplus}} & \tilde{\mu}[\iota_1\,(x).\langle \iota_1\,(x)\|e\rangle \mid \iota_1\,(y).\langle \iota_1\,(y)\|e\rangle] \\
(\eta_{\mathcal{P}}^{\otimes}) & e : A \otimes B \prec_{\eta_{\mathcal{P}}^{\otimes}} & \tilde{\mu}[(x,y).\langle (x,y)\|e\rangle] \\
(\eta_{\mathcal{P}}^{\sim}) & e : {\sim}A \prec_{\eta_{\mathcal{P}}^{\sim}} & \tilde{\mu}[{\sim}\,(\alpha).\langle {\sim}\,(\alpha)\|e\rangle]
\end{array}\right\} \quad x, y, \alpha \notin FV(e)$$

Negative $\eta_{\mathcal{P}}$ rules:

$$\left.\begin{array}{lr@{\;}l}
(\eta_{\mathcal{P}}^{\top}) & v : \top \prec_{\eta_{\mathcal{P}}^{\top}} & \mu() \\
(\eta^{\bot}) & v : \bot \prec_{\eta_{\mathcal{P}}^{\bot}} & \mu([].\langle v\|[]\rangle) \\
(\eta_{\mathcal{P}}^{\&}) & v : A \,\&\, B \prec_{\eta_{\mathcal{P}}^{\&}} & \mu(\pi_1\,[\alpha].\langle v\|\pi_1\,[\alpha]\rangle \mid \pi_2\,[\beta].\langle v\|\pi_2\,[\beta]\rangle) \\
(\eta_{\mathcal{P}}^{\parr}) & v : A \parr B \prec_{\eta_{\mathcal{P}}^{\parr}} & \mu([\alpha,\beta].\langle v\|[\alpha,\beta]\rangle) \\
(\eta_{\mathcal{P}}^{\rightarrow}) & v : A \rightarrow B \prec_{\eta_{\mathcal{P}}^{\rightarrow}} & \mu([x \cdot \beta].\langle v\|x \cdot \beta\rangle) \\
(\eta_{\mathcal{P}}^{\neg}) & v : \neg A \prec_{\eta_{\mathcal{P}}^{\neg}} & \mu(\neg\,[x].\langle v\|\neg\,[x]\rangle)
\end{array}\right\} \quad \alpha, \beta, x \notin FV(v)$$

FIGURE 4.4. The extensional η laws for system L: with two unit types (1, $\top$), two empty types (0, $\bot$), (co-)products (&, $\oplus$), (co-)pairs ($\otimes$, $\parr$), two negations ($\sim$, $\neg$), and functions ($\rightarrow$).

represented by a choice and “or” is represented by a pair because they are about refutations rather than proofs. *End remark* 4.1.

The advantage of the system L style of syntax can be seen when we look to the program transformations corresponding to the reversibility expansions previously seen in Section 4.1, which are listed in Figure 4.4. In particular, these expansions correspond to the η laws from the λ-calculus, so we refer to them by the same naming convention. For example, the expansion of the right function introduction corresponds to the λ-calculus η law for functions ($v : A \rightarrow B \prec_{\eta^{\rightarrow}} \lambda x.v\ x$) which in system L looks like:

$$(\eta_{\mathcal{P}}^{\rightarrow}) \qquad\qquad v : A \rightarrow B \prec_{\eta_{\mathcal{P}}^{\rightarrow}} \mu([x \cdot \beta].\langle v\|x \cdot \beta\rangle)$$

Here, the pattern-matching formulation of functional terms gives a more pleasant η law than the λ-based syntax from the dual calculi, which must introduce an extra output abstraction to express the $\eta_{\mathcal{P}}^{\rightarrow}$ law of the sequent calculus as follows:

$$v : A \to B \prec \lambda x.\mu\beta.\,\langle v \| x \cdot \beta \rangle$$

As another example, the expansion of the right product introduction corresponds to the surjective η law for products $(v : A \times B \prec_{\eta^{\times}} (\pi_1(v), \pi_2(v)))$ which in system L looks like:

$$(\eta_{\mathcal{P}}^{\&}) \qquad\qquad v : A \,\&\, B \prec_{\eta_{\mathcal{P}}^{\&}} \mu(\pi_1\,[\alpha].\langle v \| \pi_1\,[\alpha] \rangle \mid \pi_2\,[\beta].\langle v \| \pi_2\,[\beta] \rangle)$$

Again, the pattern-matching syntax for product terms makes for a cleaner presentation of the surjectivity of products in the sequent calculus, where the dual calculi representation of the $\eta_{\mathcal{P}}^{\&}$ introduces two output abstractions as follows:

$$v : A \times B \prec (\mu\alpha.\,\langle v \| \pi_1\,[\alpha] \rangle, \mu\beta.\,\langle v \| \pi_2\,[\beta] \rangle)$$

We also have the positive reversibility expansions which worked on the left instead of the right, meaning that they expand co-terms instead of terms. For example the left sum introduction expansion $\eta_{\mathcal{P}}^{\oplus}$ is:

$$(\eta_{\mathcal{P}}^{\oplus}) \qquad\qquad e : A \oplus B \prec_{\eta_{\mathcal{P}}^{\oplus}} \tilde{\mu}[\iota_1\,(x).\langle \iota_1\,(x) \| e \rangle \mid \iota_1\,(y).\langle \iota_1\,(y) \| e \rangle]$$

The system L η law for sums looks very different than the one we saw in the λ-calculus $(v : A + B \prec_{\eta^{+}} \mathbf{case}\, v\, \mathbf{of}\, \iota_1\,(x) \Rightarrow \iota_1\,(x) \mid \iota_2\,(y) \Rightarrow \iota_2\,(y))$. In particular, the existence of co-terms as full-fledged syntactic entities, which were missing from the syntax of the λ-calculus, gives a better presentation of the positive η laws that reveals their connection with the negative η laws. In the λ-calculus, there doesn't seem to be much connection between the η laws for sums and products, but in system L, the syntax makes it apparent that they are the polar opposite forms of the same law; one acting on terms and the other on co-terms.

Polarizing the Fundamental Dilemma

System L is a great language for expressing the extensional η laws of types in a way that reveals their symmetry with one another. However, if we try to naïvely reconcile the polarized $\eta_{\mathcal{P}}$ laws with the core $\mu\tilde{\mu}$ operational laws, we quickly run into trouble since their strength is capable of re-introducing the fundamental dilemma of computation (see Section 3.2). On the one hand, the negative $\eta_{\mathcal{P}}$ laws are incompatible with the call-by-value $\mu_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}$ laws, since $\eta_{\mathcal{P}}$ can convert any term into a $\mathcal{V}$ value. For example, if we start with the usual problematic command $\langle \mu\_.c_1 \| \tilde{\mu}\_.c_2 \rangle$, an unfortunate $\eta_{\mathcal{P}}^{\to}$ expansion can convert $\mu\_.c_1$, which is not a $\mathcal{V}$ value, into $\mu([x \cdot \beta].\langle \mu\_.c_1 \| x \cdot \beta \rangle)$, which is a $\mathcal{V}$ value. This leads to the divergent reductions:

$$c_1 \leftarrow_{\mu_{\mathcal{V}}} \langle \mu\_.c_1 \| \tilde{\mu}\_.c_2 \rangle \leftarrow_{\eta_{\mathcal{P}}^{\to}} \langle \mu([x \cdot \beta].\langle \mu\_.c_1 \| x \cdot \beta \rangle) \| \tilde{\mu}\_.c_2 \rangle \to_{\tilde{\mu}_{\mathcal{V}}} c_2$$

Therefore, for the negative $\eta_{\mathcal{P}}$ laws to make sense, the (co-)terms of negative types cannot be interpreted by the call-by-value $\mathcal{V}$ strategy. On the other hand, the positive $\eta_{\mathcal{P}}$ laws are incompatible with the call-by-name $\mu_{\mathcal{N}}\tilde{\mu}_{\mathcal{N}}$ laws, since $\eta_{\mathcal{P}}$ can convert any co-term into a $\mathcal{N}$ co-value. For example, staring from the same problematic command, an unfortunate $\eta_{\mathcal{P}}^{\otimes}$ expansion can convert $\tilde{\mu}\_.c_2$, which is not a $\mathcal{N}$ co-value, into $\tilde{\mu}[(x, y).\langle (x, y) \| \tilde{\mu}\_.c_2 \rangle]$, which is a $\mathcal{N}$ co-value. This leads to the divergent reductions:

$$c_2 \leftarrow_{\mu_{\mathcal{N}}} \langle \mu\_.c_1 \| \tilde{\mu}\_.c_2 \rangle \leftarrow_{\eta_{\mathcal{P}}^{\otimes}} \langle \mu\_.c_1 \| \tilde{\mu}[(x, y).\langle (x, y) \| \tilde{\mu}\_.c_2 \rangle] \rangle \to_{\tilde{\mu}_{\mathcal{N}}} c_1$$

Therefore, for the positive $\eta_{\mathcal{P}}$ laws to make sense, the (co-)terms of positive types cannot be interpreted by the call-by-name $\mathcal{V}$ strategy. What this means is that in the face of the polarized η laws, we cannot resolve the fundamental dilemma by just imposing a language-wide evaluation strategy once and for all as we did with the dual calculi in Chapter III, since half the $\eta_{\mathcal{P}}$ laws are incompatible with call-by-value evaluation and the other half are incompatible with call-by-name.

Fortunately, the concept of reversibility give us a different answer to the fundamental non-determinism of the classical sequent calculus that leverages the $\eta_{\mathcal{P}}$ laws instead of fighting against them, with an idea that can be traced back to Danos *et al.* (1997). The key insight is that in lieu of imposing a language-wide evaluation strategy, we can use the *type* of an interacting pair of (co-)terms in a command to figure out what evaluation strategy to use for the reduction of that particular command. So when we faced with an ambiguous command like $\langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle$, we can use the type of $\mu\alpha.c_1$ and $\tilde{\mu}x.c_2$ to tell us what the term and the co-term "really look like" (Graham-Lengrand, 2015).

For example, suppose the troublesome command is between a term and co-term of type $A \otimes B$ as in:

$$\dfrac{\dfrac{\overset{\vdots\;\mathcal{D}}{c_1 : (\Gamma \vdash \alpha : A \otimes B, \Delta)}}{\Gamma \vdash \mu\alpha.c_1 : A \otimes B \mid \Delta}\,AR \qquad \dfrac{\overset{\vdots\;\mathcal{E}}{c_2 : (\Gamma, x : A \otimes B \vdash \Delta)}}{\Gamma \mid \tilde{\mu}x.c_2 : A \otimes B \vdash \Delta}\,AL}{\langle \mu\alpha.c_1 \| \tilde{\mu}x.c_1 \rangle : (\Gamma \vdash \Delta)}\,\mathit{Cut}$$

Since we know that the left rule for $\otimes$ is reversible, we can achieve an equivalent co-term that ends with $\otimes L$:

$$\dfrac{\dfrac{\overset{\vdots\;\mathcal{D}}{c_1 : (\Gamma \vdash \alpha : A \otimes B, \Delta)}}{\Gamma \vdash \mu\alpha.c_1 : A \otimes B \mid \Delta}\,AR \qquad \dfrac{\overset{\vdots\;\mathcal{E}'}{c_2' : (\Gamma, x : A, y : B \vdash \Delta)}}{\Gamma \mid \tilde{\mu}[(x,y).c_2'] : A \otimes B \vdash \Delta}\,\otimes L}{\langle \mu\alpha.c_1 \| \tilde{\mu}[(x,y).c_2'] \rangle : (\Gamma \vdash \Delta)}\,\mathit{Cut}$$

Therefore, by employing reversibility of the typing rules, we discovered that there wasn't an issue after all, revealing the fact that in a sense the co-term was concealing its intent (Graham-Lengrand, 2015). On the other hand, if we have the command $\langle V \| \tilde{\mu}x.c \rangle$, where V is a $\mathcal{V}$ value then it is safe to substitute V for x since it must be a pair $\langle V_1 \| V_2 \rangle$ (or a variable standing in for a pair).

This approach of using reversibility of restore confluence also extends to the negative connectives. However, because negative connectives are reversible in opposite ways to positive connectives, we get the opposite resolution to the dilemma. Suppose again that we are faced with the command $\langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle$ with a similar typing derivation as before, except that now x and α have the type $A \to B$. We know that the right rule for $\to$ is reversible, so we can explicate the typing derivation as:

$$\dfrac{\dfrac{\overset{\vdots\;\mathcal{D}'}{c_1' : (\Gamma, y : A \vdash \beta : B, \Delta)}}{\Gamma \mid \mu([y \cdot \beta].c_1') : A \to B \vdash \Delta}\,{\to}R \qquad \dfrac{\overset{\vdots\;\mathcal{E}}{c_2 : (\Gamma, x : A \to B \vdash \Delta)}}{\Gamma \mid \tilde{\mu}x.c_2 : A \to B \vdash \Delta}\,AL}{\langle \mu([y \cdot \beta].c_1') \| \tilde{\mu}x.c_2 \rangle : (\Gamma \vdash \Delta)}\,\mathit{Cut}$$

giving us the more explicit command $\langle \mu([\alpha, \beta].c_1') \| \tilde{\mu}x.c_2 \rangle$ that now spells out exactly which side should be prioritized. Therefore, for negative types, polarity in the type of a cut reveals the opposite intent, restoring determinism to the system by favoring the co-term over the term. Therefore polarity of the type of a cut can tell us who requires priority, and restores determinism in an analogous manner as in Section 3.3.

Following this regime for solving the fundamental dilemma, the polarization hypothesis says that types can be used to determine the evaluation order in a program according to their polarity (Zeilberger, 2009; Munch-Maccagnoni, 2013). For positive types like $A \otimes B$ and $A \oplus B$, reversibility on the left tells us to favor giving priority to the term in case of ambiguity. Contrarily, the reversibility on the right for negative types like $A \mathbin{\&} B$, $A \parr B$, and $A \to B$ tells us to favor giving priority to the co-term. Thus, positive types suggest a call-by-value evaluation order and negative types suggest a call-by-name evaluation order. To formally apply the polarized approach to evaluation strategy, we must bifurcate the syntax of the core $\mu\tilde{\mu}$-calculus and separate the positive entities from the negative ones, as shown in Figure 4.5. This bifurcated syntax has all the same types and expressions as from Figure 3.7, except that positive types and (co-)terms (denoted by $A_+, \dots, v_+$, and e_+) are syntactically separate from negative types and (co-)terms (denoted by $A_-, \dots, v_-$, and e_-). In order for the polarity of a type or (co-)term to be apparent from its syntax, we need to annotate type variables and (co-)variables with their intended polarity using either the positive superscript (X^+, x^+, α^+) or the negative superscript (X^-, x^-, α^-) which is an explicit part of their syntax (as opposed to the mere distinction between v_+ and v_-, etc.). When the polarity of type, term, co-term doesn't matter, we may just refer to it as A for either A_+ or A_-, v for either an v_+ or v_-, and e for either a e_+ or e_-. Note commands are not distinguished by a polarity because unlike (co-)terms they are not part of a specific type. Instead, the single syntactic set *Command* contains two different kinds of commands—one between positive (co-)terms and one between negative ones—so that commands are only syntactically valid when the polarity of their (co-)terms agree. Also, note that only the core typing rules are bifurcated into positive and negative versions; the structural rules from Figure 3.10 which are also part of $\mu\tilde{\mu}_{\mathcal{P}}$ remain the same.

Now, to address polarity in the full system L language, we only need to extend the polarized core $\mu\tilde{\mu}_{\mathcal{P}}$-calculus with the specific connectives and constructs, as shown in Figure 4.6, which extends the polarized core calculus from Figure 4.5. Note that there is one extra pair of connectives $\downarrow A_-$ and $\uparrow A_+$ that are introduced in Figure 4.6 which are known as Girard's (2001) polarity "shifts" that mark a switch between the positive and negative polarities. These shifts are important for making sure that

$$A, B, C \in \mathit{Type} ::= A_+ \mid A_-$$

$$A_+, B_+, C_+ \in \mathit{Type}_+ ::= X^+ \qquad\qquad A_-, B_-, C_- \in \mathit{Type}_- ::= X^-$$

$$c \in \mathit{Command} ::= \langle v_+ \| e_+ \rangle \mid \langle v_- \| e_- \rangle$$

$$v \in \mathit{Term} ::= v_+ \mid v_- \qquad v_+ \in \mathit{Term}_+ ::= x^+ \mid \mu\alpha^+.c \qquad v_- \in \mathit{Term}_- ::= x^- \mid \mu\alpha^-.c$$

$$e \in \mathit{CoTerm} ::= e_+ \mid e_- \quad e_+ \in \mathit{CoTerm} ::= \alpha^+ \mid \tilde{\mu}x^+.c \quad e_- \in \mathit{CoTerm} ::= \alpha^- \mid \tilde{\mu}x^-.c$$

$$\Gamma \in \mathit{InputEnv} ::= x_1 : A_1, \dots, x_n : A_n \qquad \Delta \in \mathit{OutputEnv} ::= \alpha_1 : A_1, \dots, \alpha_n : A_n$$

$$\mathit{Judgement} ::= c : (\Gamma \vdash \Delta) \mid (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta)$$

Core rules:

$$\frac{}{x^+ : A_+ \vdash x^+ : A_+ \mid}\ \mathit{VR}_+ \qquad \frac{}{\mid \alpha^+ : A_+ \vdash \alpha^+ : A_+}\ \mathit{VL}_+$$

$$\frac{}{x^- : A_- \vdash x^- : A_- \mid}\ \mathit{VR}_- \qquad \frac{}{\mid \alpha^- : A_- \vdash \alpha^- : A_-}\ \mathit{VL}_-$$

$$\frac{c : (\Gamma \vdash \alpha^+ : A_+, \Delta)}{\Gamma \vdash \mu\alpha^+.c : A_+ \mid \Delta}\ \mathit{AR}_+ \qquad \frac{c : (\Gamma, x^+ : A_+ \vdash \Delta)}{\Gamma \mid \tilde{\mu}x^+.c : A_+ \vdash \Delta}\ \mathit{AL}_+$$

$$\frac{c : (\Gamma \vdash \alpha^- : A_-, \Delta)}{\Gamma \vdash \mu\alpha^-.c : A_- \mid \Delta}\ \mathit{AR}_- \qquad \frac{c : (\Gamma, x^- : A_- \vdash \Delta)}{\Gamma \mid \tilde{\mu}x^-.c : A_- \vdash \Delta}\ \mathit{AL}_-$$

$$\frac{\Gamma \vdash v_+ : A_+ \mid \Delta \quad \Gamma' \mid e_+ : A_+ \vdash \Delta'}{\langle v_+ \| e_+ \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)}\ \mathit{Cut}_+ \qquad \frac{\Gamma \vdash v_- : A_- \mid \Delta \quad \Gamma' \mid e_- : A_- \vdash \Delta'}{\langle v_- \| e_- \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)}\ \mathit{Cut}_-$$

$$V \in \mathit{Value}_{\mathcal{P}} ::= V_+ \mid V_- \qquad V_+ \in \mathit{Value}_+ ::= x^+ \qquad V_- \in \mathit{Value}_- ::= v_-$$

$$E \in \mathit{CoValue}_{\mathcal{P}} ::= E_+ \mid E_- \quad E_+ \in \mathit{CoValue}_+ ::= e_+ \quad E_- \in \mathit{CoValue}_- ::= \alpha^-$$

$$\begin{array}{llll}
(\mu_{\mathcal{P}}) & \left\langle \mu\alpha^+.c \middle\| E_+ \right\rangle \succ_{\mu_{\mathcal{P}}} c\left\{E_+/\alpha^+\right\} & (\eta_\mu) & \mu\alpha^+.\left\langle v_+ \middle\| \alpha^+ \right\rangle \succ_{\eta_\mu} v_+ \quad (\alpha^+ \notin FV(v_+)) \\
(\mu_{\mathcal{P}}) & \left\langle \mu\alpha^-.c \middle\| E_- \right\rangle \succ_{\mu_{\mathcal{P}}} c\left\{E_-/\alpha^-\right\} & (\eta_\mu) & \mu\alpha^-.\left\langle v_- \middle\| \alpha^- \right\rangle \succ_{\eta_\mu} v_- \quad (\alpha^- \notin FV(v_-)) \\
(\tilde{\mu}_{\mathcal{P}}) & \left\langle V_+ \middle\| \tilde{\mu}x^+.c \right\rangle \succ_{\mu_{\mathcal{P}}} c\left\{V_+/x^+\right\} & (\eta_\mu) & \tilde{\mu}x^+.\left\langle x^+ \middle\| e_+ \right\rangle \succ_{\eta_\mu} e_+ \quad (x^+ \notin FV(e_+)) \\
(\tilde{\mu}_{\mathcal{P}}) & \left\langle V_- \middle\| \tilde{\mu}x^-.c \right\rangle \succ_{\mu_{\mathcal{P}}} c\left\{V_-/x^-\right\} & (\eta_\mu) & \tilde{\mu}x^-.\left\langle x^- \middle\| e_- \right\rangle \succ_{\eta_\mu} e_- \quad (x^- \notin FV(e_-))
\end{array}$$

FIGURE 4.5. The polarized core $\mu\tilde{\mu}_{\mathcal{P}}$-calculus: its static and dynamic semantics.

$$
\begin{aligned}
A, B, C \in \mathit{Type} &::= A_+ \mid A_- \\
A_+, B_+, C_+ \in \mathit{Type}_+ &::= X^+ \mid 0 \mid 1 \mid A_+ \oplus B_+ \mid A_+ \otimes B_+ \mid {\sim} A_- \mid {\downarrow} A_- \\
A_-, B_-, C_- \in \mathit{Type}_- &::= X^- \mid \top \mid \bot \mid A_- \mathbin{\&} B_- \mid A_- \parr B_- \mid A_+ \to B_- \mid \neg A_+ \mid {\uparrow} A_+
\end{aligned}
$$

$$
c \in \mathit{Command} ::= \langle v_+ \| e_+ \rangle \mid \langle v_- \| e_- \rangle \quad v \in \mathit{Term} ::= v_+ \mid v_- \quad e \in \mathit{CoTerm} ::= e_+ \mid e_-
$$

$$
\begin{aligned}
v_+ \in \mathit{Term}_+ &::= x^+ \mid \mu\alpha^+.c \mid () \mid \iota_1\,(v_+) \mid \iota_2\,(v_+) \mid (v_+, v_+) \mid {\sim}\,(e_-) \mid {\downarrow}(v_-) \\
e_+ \in \mathit{CoTerm}_+ &::= \alpha^+ \mid \tilde{\mu}x^+.c \mid \tilde{\mu}[] \mid \tilde{\mu}[().c] \mid \tilde{\mu}\Big[\iota_1\left(x^+\right).c \mid \iota_2\left(y^+\right).c\Big] \\
&\quad \mid \tilde{\mu}\Big[\Big(x^+, y^+\Big).c\Big] \mid \tilde{\mu}\Big[{\sim}\left(\alpha^-\right).c\Big] \mid \tilde{\mu}\Big[{\downarrow}\Big(x^-\Big).c\Big]
\end{aligned}
$$

$$
\begin{aligned}
v_- \in \mathit{Term}_- &::= x^- \mid \mu\alpha^-.c \mid \mu() \mid \mu([].c) \mid \mu\Big(\pi_1\left[\alpha^-\right].c \mid \pi_2\left[\beta^-\right].c\Big) \\
&\quad \mid \mu\Big(\Big[\alpha^-, \beta^-\Big].c\Big) \mid \mu\Big([x^+ \cdot \alpha^-].c\Big) \mid \mu\Big(\neg\left[x^+\right].c\Big) \mid \mu\Big({\uparrow}\Big[\alpha^+\Big].c\Big) \\
e_- \in \mathit{CoTerm}_- &::= \alpha^- \mid \tilde{\mu}x^-.c \mid [] \mid \pi_1\,[e_-] \mid \pi_2\,[e_-] \mid [e_-, e_-] \mid v_+ \cdot e_- \mid \neg\,[v_+] \mid {\uparrow}[e_+]
\end{aligned}
$$

FIGURE 4.6. The syntax for polarized system L: with both positive connectives—disjunction ($\oplus$), conjunction ($\otimes$), negation ($\sim$), and polarity shift ($\downarrow$)—and negative connectives—conjunction ($\&$), disjunction ($\parr$), negation ($\neg$), functions ($\to$), and polarity shift ($\uparrow$).

the polar bifurcation of the language does not accidentally eliminate its essential expressive capabilities. For example, in the λ-calculus, the dual calculi, or a functional programming language, it is typical to store a function (which is a negative term) inside the structure of a pair or sum type structure (which is a positive term). However, this would be prevented by the distinction between the polarities of types. Instead, we would like to allow for some mingling between positive and negative types and (co-)terms without confusing the two. The $\downarrow$ shift lets us embed negative types inside positive ones, so that for every negative type A_- we have the positive type $\downarrow A_-$. Going along with our story that positive values follow predetermined patterns, we have the structured term $\downarrow(v_-) : \downarrow A_-$ which contains a negative term along with a case abstraction co-term, $\tilde{\mu}[\downarrow(x^-).c] : \downarrow A_-$, for unpacking the structure and pulling out the underlying term. The $\uparrow$ shift lets us embed positive types inside negative ones, so that for every type A_+ we have the negative type $\uparrow A_+$. The (co-)terms of the $\uparrow$ shift symmetric to the $\downarrow$ ones, so that we have the co-case abstraction term $\mu(\uparrow[\alpha^+].c) : \uparrow A_+$ which is waiting for a shifted co-term of the form $\uparrow[e_+] : \uparrow A_+$ containing a positive co-term.

The logical typing rules for polarized system L is shown in Figure 4.7, which are effectively the same rules from Figure 4.3 made aware of the distinction between positive and negative polarities. The only new rules are for the new shift connectives. More interestingly, the $\beta_{\mathcal{P}}$ rules for system L are similar to rules for reducing case analysis in functional languages as shown in Figure 4.8. For example, for the positive $\beta_{\mathcal{P}}$ laws we have sum types that select which branch to take based on the constructor tag:

$$\left\langle \iota_1\left(V_+\right) \middle\| \tilde{\mu}\left[\iota_1\left(x^+\right).c_1 \mid \iota_2\left(y^+\right).c_2\right]\right\rangle \succ_{\beta_{\mathcal{P}}^{\oplus}} c_1\left\{V_+/x^+\right\}$$

and pair types which decompose a pair into its constituent parts:

$$\left\langle \left(V_+, V'_+\right) \middle\| \tilde{\mu}\left[\left(x^+, y^+\right).c\right]\right\rangle \succ_{\beta_{\mathcal{P}}^{\otimes}} c\left\{V_+/x^+, V'_+/y^+\right\}$$

The negative $\beta[][\mathcal{P}]$ laws follow the same notion of case analysis as the positive $\beta[][\mathcal{P}]$ laws, except in the reverse direction. For example, terms of product types select the appropriate response based on the constructor tag of their observation:

$$\left\langle \mu\left(\pi_1\left[\alpha^-\right].c_1 \mid \pi_2\left[\beta^-\right].c_2\right) \middle\| \pi_1\left[E_-\right]\right\rangle \succ_{\beta_{\mathcal{P}}^{\&}} c_1\left\{E_-/\alpha^-\right\}$$

Positive logical rules:

no $0R$ rule $$\frac{}{\Gamma \mid \tilde{\mu}[] : 0 \vdash \Delta}\ 0L \qquad \frac{}{\vdash () : 1 \mid}\ 1R \qquad \frac{c : (\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}[().c] : 1 \vdash \Delta}\ 1L$$

$$\frac{\Gamma \vdash v_+ : A_+ \mid \Delta}{\Gamma \vdash \iota_1\,(v_+) : A_+ \oplus B_+ \mid \Delta}\ \oplus R_1 \qquad \frac{\Gamma \vdash v_+ : B_+ \mid \Delta}{\Gamma \vdash \iota_2\,(v_+) : A_+ \oplus B_+ \mid \Delta}\ \oplus R_2$$

$$\frac{c : (\Gamma, x^+ : A_+ \vdash \Delta) \quad c' : (\Gamma, y^+ : B_+ \vdash \Delta)}{\Gamma \mid \tilde{\mu}[\iota_1\,(x^+).c \mid \iota_2\,(y^+).c'] : A_+ \oplus B_+ \vdash \Delta}\ \oplus L$$

$$\frac{\Gamma \vdash v_+ : A_+ \mid \Delta \quad \Gamma' \vdash v'_+ : B_+ \mid \Delta'}{\Gamma, \Gamma' \vdash \left(v_+, v'_+\right) : A_+ \otimes B_+ \mid \Delta, \Delta'}\ \otimes R \qquad \frac{c : (\Gamma, x^+ : A_+, y^+ : B_+ \vdash \Delta)}{\Gamma \mid \tilde{\mu}[(x^+, y^+).c] : A_+ \otimes B_+ \vdash \Delta}\ \otimes L$$

$$\frac{\Gamma \mid e_- : A_- \vdash \Delta}{\Gamma \vdash \sim(e_-) : \sim A_- \mid \Delta}\ \sim R \qquad \frac{c : (\Gamma \vdash \alpha^- : A_-, \Delta)}{\Gamma \mid \tilde{\mu}[\sim(\alpha^-).c] : \sim A_- \vdash \Delta}\ \sim L$$

$$\frac{\Gamma \vdash v_- : A_- \mid \Delta}{\Gamma \vdash {\downarrow}(v_-) : {\downarrow}A_- \mid \Delta}\ {\downarrow}R \qquad \frac{c : (\Gamma, x^- : A_- \vdash \Delta)}{\Gamma \mid \tilde{\mu}[{\downarrow}(x^-).c] : {\downarrow}A_- \vdash \Delta}\ {\downarrow}L$$

Negative logical rules:

$$\frac{}{\Gamma \vdash \mu() : \top \mid \Delta}\ \top R$$ no $\top L$ rule $$\frac{c : (\Gamma \vdash \Delta)}{\Gamma \vdash \mu([].c) : \bot \mid \Delta}\ \bot R \qquad \frac{}{\mid [] : \bot \vdash}\ \bot L$$

$$\frac{c : (\Gamma \vdash \alpha^- : A_-, \Delta) \quad c' : (\Gamma \vdash \beta^- : B_-, \Delta)}{\Gamma \vdash \mu(\pi_1\,[\alpha^-].c \mid \pi_2\,[\beta^-].c') : A_- \,\&\, B_- \mid \Delta}\ \&R$$

$$\frac{\Gamma \mid e_- : A_- \vdash \Delta}{\Gamma \mid \pi_1\,[e_-] : A_- \,\&\, B_- \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma \mid e_- : B_- \vdash \Delta}{\Gamma \mid \pi_2\,[e_-] : A_- \,\&\, B_- \vdash \Delta}\ \&L_2$$

$$\frac{c : (\Gamma \vdash \alpha^- : A_-, \beta^- : B_-, \Delta)}{\Gamma \vdash \mu([\alpha^-, \beta^-].c) : A_- \parr B_- \mid \Delta}\ \parr R \qquad \frac{\Gamma \mid e_- : A_- \vdash \Delta \quad \Gamma' \mid e'_- : B_- \vdash \Delta'}{\Gamma, \Gamma' \mid \left[e_-, e'_-\right] : A_- \parr B_- \vdash \Delta, \Delta'}\ \parr L$$

$$\frac{c : (\Gamma, x^+ : A_+ \vdash \beta^- : B_-, \Delta)}{\Gamma \vdash \mu(x^+ \cdot \beta^-.c) : A_+ \to B_- \mid \Delta}\ {\to}R \qquad \frac{\Gamma \vdash v_+ : A_+ \mid \Delta \quad \Gamma' \mid e_- : B_- \vdash \Delta'}{\Gamma, \Gamma' \mid v_+ \cdot e_- : A_+ \to B_- \vdash \Delta, \Delta'}\ {\to}L$$

$$\frac{c : (\Gamma, x^+ : A_+ \vdash \Delta)}{\Gamma \vdash \mu(\neg\,[x^+].c) : \neg A_+ \mid \Delta}\ \neg R \qquad \frac{\Gamma \vdash v_+ : A_+ \mid \Delta}{\Gamma \mid \neg\,[v_+] : \neg A_+ \vdash \Delta}\ \neg L$$

$$\frac{c : (\Gamma \vdash \alpha^+ : A_+, \Delta)}{\Gamma \vdash \mu({\uparrow}[\alpha^+].c) : {\uparrow}A_+ \mid \Delta}\ {\uparrow}R \qquad \frac{\Gamma \mid e_+ : A_+ \vdash \Delta}{\Gamma \mid {\uparrow}[e_+] : {\uparrow}A_+ \vdash \Delta}\ {\uparrow}L$$

FIGURE 4.7. Logical typing rules for polarized system L: with both positive connectives (0, 1, $\oplus$, $\otimes$, $\sim$, $\downarrow$) and negative connectives ($\top$, $\bot$, $\&$, $\parr$, $\to$, $\neg$, $\uparrow$).

$$
\begin{aligned}
V \in Value_{\mathcal{P}} &::= V_+ \mid v_- \\
V_+ \in Value_+ &::= x^+ \mid () \mid \iota_1\left(V_+\right) \mid \iota_2\left(V_+\right) \mid \left(V_+, V_+\right) \mid \sim\left(E_-\right) \mid {\downarrow}(v_-) \\
E \in CoValue_{\mathcal{P}} &::= e_+ \mid E_- \\
E_- \in CoValue_- &::= \alpha^- \mid \pi_1\left[E_-\right] \mid \pi_2\left[E_-\right] \mid \left[E_-, E_-\right] \mid V_+ \cdot E_- \mid \neg\left[V_+\right] \mid {\uparrow}[e_+]
\end{aligned}
$$

Positive $\beta_{\mathcal{P}}$ rules:

$$
\begin{array}{ll}
(\beta^0_{\mathcal{P}}) & \text{no } \beta^0_{\mathcal{P}} \text{ rule} \\
(\beta^1_{\mathcal{P}}) & \langle () \| \tilde{\mu}[().c] \rangle \succ_{\beta^1_{\mathcal{P}}} c \\
(\beta^{\oplus}_{\mathcal{P}}) & \Big\langle \iota_i\left(V_+\right) \Big\| \tilde{\mu}\Big[\iota_1\left(x_1^+\right).c_1 \mid \iota_2\left(x_2^+\right).c_2\Big] \Big\rangle \succ_{\beta^{\oplus}_{\mathcal{P}}} c_i \left\{V_+/x_i^+\right\} \\
(\beta^{\otimes}_{\mathcal{P}}) & \Big\langle \left(V_+, V'_+\right) \Big\| \tilde{\mu}\Big[\left(x^+, y^+\right).c\Big] \Big\rangle \succ_{\beta^{\otimes}_{\mathcal{P}}} c \left\{V_+/x, V'_+/y\right\} \\
(\beta^{\sim}_{\mathcal{P}}) & \Big\langle \sim\left(E_-\right) \Big\| \tilde{\mu}\Big[\sim\left(\alpha^-\right).c\Big] \Big\rangle \succ_{\beta^{\sim}_{\mathcal{P}}} c \left\{E_-/\alpha^-\right\} \\
(\beta^{\downarrow}_{\mathcal{P}}) & \Big\langle {\downarrow}(v_-) \Big\| \tilde{\mu}\Big[{\downarrow}\left(x^-\right).c\Big] \Big\rangle \succ_{\beta^{\downarrow}_{\mathcal{P}}} c \left\{v_-/x^-\right\}
\end{array}
$$

Negative $\beta_{\mathcal{P}}$ rules:

$$
\begin{array}{ll}
(\beta^{\top}_{\mathcal{P}}) & \text{no } \beta^{\top}_{\mathcal{P}} \text{ rule} \\
(\beta^{\bot}_{\mathcal{P}}) & \langle \mu([].c) \| [] \rangle \succ_{\beta^{\bot}_{\mathcal{P}}} c \\
(\beta^{\&}_{\mathcal{P}}) & \Big\langle \mu\Big(\pi_1\left[\alpha_2^-\right].c_1 \mid \pi_2\left[\alpha_1^-\right].c_2\Big) \Big\| \pi_i\left[E_-\right] \Big\rangle \succ_{\beta^{\&}_{\mathcal{P}}} c_i \left\{E_-/\alpha_i^-\right\} \\
(\beta^{\parr}_{\mathcal{P}}) & \Big\langle \mu\Big(\left[\alpha^-, \beta^-\right].c\Big) \Big\| \left[E_-, E'_-\right] \Big\rangle \succ_{\beta^{\parr}_{\mathcal{P}}} c \left\{E_-/\alpha^-, E'_-/\beta^-\right\} \\
(\beta^{\to}_{\mathcal{P}}) & \Big\langle \mu\Big([x^+ \cdot \beta^-].c\Big) \Big\| V_+ \cdot E_- \Big\rangle \succ_{\beta^{\to}_{\mathcal{P}}} c \left\{V_+/x^+, E_-/\beta^-\right\} \\
(\beta^{\neg}_{\mathcal{P}}) & \Big\langle \mu\Big(\neg\left[x^+\right].c\Big) \Big\| \neg\left[V_+\right] \Big\rangle \succ_{\beta^{\neg}_{\mathcal{P}}} c \left\{V_+/x^+\right\} \\
(\beta^{\uparrow}_{\mathcal{P}}) & \Big\langle \mu\Big({\uparrow}\left[\alpha^+\right].c\Big) \Big\| {\uparrow}[e_+] \Big\rangle \succ_{\beta^{\uparrow}_{\mathcal{P}}} c \left\{e_+/\alpha^+\right\}
\end{array}
$$

FIGURE 4.8. The operational β laws for polarized system L: with two unit types (1, $\top$), two empty types (0, $\bot$), (co-)products ($\&$, $\oplus$), (co-)pairs ($\otimes$, $\parr$), two negations ($\sim$, $\neg$), functions ($\to$), and two polarity shifts ($\downarrow$, $\uparrow$).

and terms of co-pair types decompose their observation into the two independent messages:

$$\left\langle \mu\left(\left[\alpha^-, \beta^-\right].c\right) \middle\| \left[E_-, E'_-\right] \right\rangle \succ_{\beta_{\mathcal{P}}^{\parr}} c\left\{E_-/\alpha^-, E'_-/\beta^-\right\}$$

Focusing and Polarity

The $\beta_{\mathcal{P}}$-based operational rules for polarized system L explain how to reduce commands by performing pattern-matching. However, $\beta_{\mathcal{P}}$ reduction alone is not enough, since it suffers the same essential deficiency as β reduction in the dual calculi (Section 3.3). For example, in the positive form of pattern-matching of type $A_+ \oplus B_+$, we could encounter the command

$$\left\langle \iota_1\left(\mu\alpha^+.c\right) \middle\| \tilde{\mu}\left[\iota_1\left(x^+\right).c_1 \mid \iota_2\left(y^+\right).c_2\right] \right\rangle \not\succ_{\beta_{\mathcal{P}}^{\oplus}}$$

which does not proceed by $\beta_{\mathcal{P}}^{\oplus}$ because $\mu\alpha^+.c$ is not a $\mathcal{V}$ value. Similarly in the negative form of pattern-matching of type $A_+ \to B_-$, we could encounter the command

$$\left\langle \mu\left([x^+ \cdot \beta^-].c\right) \middle\| (\mu\alpha^+.c_1) \cdot [\tilde{\mu}y^-.c_2] \right\rangle \not\succ_{\beta_{\mathcal{P}}^{\to}}$$

which does not proceed by $\beta_{\mathcal{P}}^{\to}$ because $\mu\alpha^+.c_1$ is not a $\mathcal{V}$ value and $\tilde{\mu}y^-.c_2$ is not a $\mathcal{N}$ co-value.

Unsurprisingly, the same technique of *focusing* with the same two options we had before: we can remove the superfluous parts of the syntax of system L (like the above two commands) with the static approach to focusing, or we can add the extra steps necessary to kick-start the computation again with the dynamic approach to focusing. The major difference between focusing in system L versus focusing in the dual calculi is that since polarized system L incorporates aspects of both the call-by-value and call-by-name halves of the dual calculi into a single language, the polarized focusing shares similarities with both the call-by-value and call-by-name focusing at once. In particular, the dual calculi had two different sets of focused sub-syntaxes (LKQ and LKT) and two different sets of focusing ς rules ($\varsigma_{\mathcal{V}}$ and $\varsigma_{\mathcal{N}}$) corresponding to its two different evaluation strategies. Instead, polarized system L has a single focused sub-syntax and a single set of focusing ς rules.

First, let's consider the static approach with the focused sub-syntax of system L shown in Figure 4.9. On the positive side, the restrictions on the syntax of positive terms resembles LKQ. Every positive term is either a positive value or output

$$
\begin{aligned}
v_+ \in \mathit{Term}_+ &::= V_+ \mid \mu\alpha^+.c \\
V_+ \in \mathit{Value}_+ &::= x^+ \mid () \mid \iota_1\,(V_+) \mid \iota_2\,(V_+) \mid (V_+, V_+) \mid {\sim}\,(E_-) \mid {\downarrow}(v_-) \\
e_+ \in \mathit{CoTerm}_+ &::= \alpha^+ \mid \tilde{\mu}x^+.c \mid \tilde{\mu}[] \mid \tilde{\mu}[().c] \mid \tilde{\mu}\Big[\iota_1\left(x^+\right).c \mid \iota_2\left(y^+\right).c\Big] \\
&\quad \mid \tilde{\mu}\Big[\left(x^+, y^+\right).c\Big] \mid \tilde{\mu}\Big[{\sim}\left(\alpha^-\right).c\Big] \mid \tilde{\mu}\Big[{\downarrow}\left(x^-\right).c\Big]
\end{aligned}
$$

$$
\begin{aligned}
v_- \in \mathit{Term}_- &::= x^- \mid \mu\alpha^-.c \mid \mu() \mid \mu([].c) \mid \mu\Big(\pi_1\left[\alpha^-\right].c \mid \pi_2\left[\beta^-\right].c\Big) \\
&\quad \mid \mu\Big(\left[\alpha^-, \beta^-\right].c\Big) \mid \mu\Big([x^+ \cdot \alpha^-].c\Big) \mid \mu\Big(\neg\left[x^+\right].c\Big) \mid \mu\Big({\uparrow}\left[\alpha^+\right].c\Big) \\
e_- \in \mathit{CoTerm}_- &::= E_- \mid \tilde{\mu}x^-.c \\
E_- \in \mathit{CoValue}_- &::= \alpha^- \mid \pi_1\,[E_-] \mid \pi_2\,[E_-] \mid [E_-, E_-] \mid V_+ \cdot E_- \mid \neg\,[V_+] \mid {\uparrow}[e_+]
\end{aligned}
$$

$$
\begin{aligned}
c \in \mathit{Command} &::= \langle v_+ \| e_+ \rangle \mid \langle v_- \| e_- \rangle \\
\mathit{Judgement} &::= c : (\Gamma \vdash \Delta) \\
&\quad \mid (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \vdash V_+ : A_+ \,;\, \Delta) \\
&\quad \mid (\Gamma \mid e : A \vdash \Delta) \mid (\Gamma \,;\, E_- : A_- \vdash \Delta)
\end{aligned}
$$

Axiom:

$$
\frac{}{x^+ : A_+ \vdash x^+ : A_+ \,;}\ \mathit{Var}^+
\qquad\qquad
\frac{}{\mid \alpha^+ : A_+ \vdash \alpha^+ : A_+}\ \mathit{CoVar}^+
$$

$$
\frac{}{x^- : A_- \vdash x^- : A_- \mid}\ \mathit{Var}^-
\qquad\qquad
\frac{}{;\, \alpha^- : A_- \vdash \alpha^- : A_-}\ \mathit{CoVar}^-
$$

Focusing (structural) rules:

$$
\frac{\Gamma \vdash V_+ : A_+ \,;\, \Delta}{\Gamma \vdash V_+ : A_+ \mid \Delta}\ \mathit{FR}
\qquad\qquad
\frac{\Gamma \,;\, E_- : A_- \vdash \Delta}{\Gamma \mid E_- : A_- \vdash \Delta}\ \mathit{FL}
$$

FIGURE 4.9. Focused sub-syntax and core typing rules for polarized system L.

abstraction, where the positive values are defined hereditarily: a pair of two values is a value, an injection of a value is a value, and so on. That way, troublesome commands like $\langle \iota_1\,(\mu\alpha^+.c) \| \tilde{\mu}[\iota_1\,(x^+).c_1 \mid \iota_2\,(y^+).c_2] \rangle$ become syntactically forbidden. The interesting types that contain negative types and break this mold are the values $\sim(E_-) : {\sim}A_-$ which contain a negative co-value and the values $\downarrow(v_-) : \downarrow A_-$ which contains a negative term. Also like LKQ there is no restrictions placed on positive co-terms, which is in part because the co-terms of positive types are all abstractions which are not easily restricted like the positively constructed terms are. On the negative side, the restrictions on the syntax of negative co-terms resembles LKT. Every negative term is either a negative co-value or input abstraction, where the negative co-values are defined hereditarily: a pair of two co-value is a co-value, a projection of a co-value is a co-value, etc. So troublesome commands like $\langle \mu([x^+ \cdot \beta^-].c) \| (\mu\alpha^+.c_1) \cdot [\tilde{\mu}y^-.c_2] \rangle$ are also syntactically forbidden. As before, there are some interesting types that refer to positive types, like the co-values $V_+ \cdot E_- : A_+ \to B_-$ and $\neg\,[V_+] : \neg A_+$ which contain a positive value and $\uparrow[e_+] : \uparrow A_+$ which contains a positive co-term. Also as like LKT there is no restriction on the negative terms, which are all abstractions over negative co-values.

The focalized and polarized type system for system L introduces two new sequents using the stoup (;) based on the two restrictions on the syntax, $\Gamma \vdash V_+ : A_+ \,;\, \Delta$ for typing positive values in focus and $\Gamma \,;\, E_- : A_- \vdash \Delta$ for typing negative co-values in focus. The logical typing rules are given in Figure 4.10. The typing rules are essentially the same as the unfocused polarized ones from Figure 4.7, except that they now follow the syntactic restrictions on positive terms and negative co-terms Figure 4.9. This has the net effect that, in a bottom-up reading of a typing derivation, once focus is gained via the *FR* or *FL* rules it is maintained. The *only* rules which are capable of losing focus are the $\downarrow R$ and $\uparrow L$ rules, which transition from a positive value to a negative term and from a negative co-value to a positive co-term. This can be seen as a design philosophy justifying the choice of polarities in the connectives of polarized system L from Figure 4.6: focus should be maintained by every connective *except* the shifts. Therefore, the function type $A_+ \to B_-$ (called the "primordial function type" by Zeilberger (2009)) must have a positive argument type and negative return type to maintain focus in the call stack $V_+ \cdot E_-$, and the negation types ${\sim}A_-$ and $\neg A_+$ must invert the polarity of the type to maintain focus in $\sim(E_-)$ and $\neg\,[V_+]$. Anything else

Positive focused logical rules:

no $0R$ rule $\qquad \dfrac{}{\Gamma \mid \tilde{\mu}[] : 0 \vdash \Delta}\ 0L \qquad \dfrac{}{\vdash () : 1\ ;}\ 1R \qquad \dfrac{c : (\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}[().c] : 1 \vdash \Delta}\ 1L$

$$\frac{\Gamma \vdash V_+ : A_+ \ ;\ \Delta}{\Gamma \vdash \iota_1\,(V_+) : A_+ \oplus B_+ \ ;\ \Delta}\ \oplus R_1 \qquad \frac{\Gamma \vdash V_+ : B_+ \ ;\ \Delta}{\Gamma \vdash \iota_2\,(V_+) : A_+ \oplus B_+ \ ;\ \Delta}\ \oplus R_2$$

$$\frac{c : (\Gamma, x^+ : A_+ \vdash \Delta) \quad c' : (\Gamma, y^+ : B_+ \vdash \Delta)}{\Gamma \mid \tilde{\mu}[\iota_1\,(x^+).c \mid \iota_2\,(y^+).c'] : A_+ \oplus B_+ \vdash \Delta}\ \oplus L$$

$$\frac{\Gamma \vdash V_+ : A_+ \ ;\ \Delta \quad \Gamma' \vdash V'_+ : B_+ \ ;\ \Delta'}{\Gamma, \Gamma' \vdash \left(V_+, V'_+\right) : A_+ \otimes B_+ \ ;\ \Delta, \Delta'}\ \otimes R \qquad \frac{c : (\Gamma, x^+ : A_+, y^+ : B_+ \vdash \Delta)}{\Gamma \mid \tilde{\mu}[(x^+, y^+).c] : A_+ \otimes B_+ \vdash \Delta}\ \otimes L$$

$$\frac{\Gamma \ ;\ E_- : A_- \vdash \Delta}{\Gamma \vdash {\sim}\,(E_-) : {\sim}A_- \ ;\ \Delta}\ {\sim}R \qquad \frac{c : (\Gamma \vdash \alpha^- : A_-, \Delta)}{\Gamma \mid \tilde{\mu}[{\sim}\,(\alpha^-).c] : {\sim}A_- \vdash \Delta}\ {\sim}L$$

$$\frac{\Gamma \vdash v_- : A_- \mid \Delta}{\Gamma \vdash {\downarrow}(v_-) : {\downarrow}A_- \ ;\ \Delta}\ {\downarrow}R \qquad \frac{c : (\Gamma, x^- : A_- \vdash \Delta)}{\Gamma \mid \tilde{\mu}[{\downarrow}(x^-).c] : {\downarrow}A_- \vdash \Delta}\ {\downarrow}L$$

Negative focused logical rules:

$\dfrac{}{\Gamma \vdash \mu() : \top \mid \Delta}\ \top R \qquad$ no $\top L$ rule $\qquad \dfrac{c : (\Gamma \vdash \Delta)}{\Gamma \vdash \mu([].c) : \bot \mid \Delta}\ \bot R \qquad \dfrac{}{;\ [] : \bot \vdash}\ \bot L$

$$\frac{c : (\Gamma \vdash \alpha^- : A_-, \Delta) \quad c' : (\Gamma \vdash \beta^- : B_-, \Delta)}{\Gamma \vdash \mu(\pi_1\,[\alpha^-].c \mid \pi_2\,[\beta^-].c') : A_- \mathbin{\&} B_- \mid \Delta}\ \&R$$

$$\frac{\Gamma \ ;\ E_- : A_- \vdash \Delta}{\Gamma \ ;\ \pi_1\,[E_-] : A_- \mathbin{\&} B_- \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma \ ;\ E_- : B_- \vdash \Delta}{\Gamma \ ;\ \pi_2\,[E_-] : A_- \mathbin{\&} B_- \vdash \Delta}\ \&L_2$$

$$\frac{c : (\Gamma \vdash \alpha^- : A_-, \beta^- : B_-, \Delta)}{\Gamma \vdash \mu([\alpha^-, \beta^-].c) : A_- \parr B_- \mid \Delta}\ \parr R \qquad \frac{\Gamma \ ;\ E_- : A_- \vdash \Delta \quad \Gamma' \ ;\ E'_- : B_- \vdash \Delta'}{\Gamma, \Gamma' \ ;\ \left[E_-, E'_-\right] : A_- \parr B_- \vdash \Delta, \Delta'}\ \parr L$$

$$\frac{c : (\Gamma, x^+ : A_+ \vdash \beta^- : B_-, \Delta)}{\Gamma \vdash \mu(x^+ \cdot \beta^-.c) : A_+ \to B_- \mid \Delta}\ {\to}R \qquad \frac{\Gamma \vdash V_+ : A_+ \ ;\ \Delta \quad \Gamma' \ ;\ E_- : B_- \vdash \Delta'}{\Gamma, \Gamma' \ ;\ V_+ \cdot E_- : A_+ \to B_- \vdash \Delta, \Delta'}\ {\to}L$$

$$\frac{c : (\Gamma, x^+ : A_+ \vdash \Delta)}{\Gamma \vdash \mu(\neg\,[x^+].c) : \neg A_+ \mid \Delta}\ \neg R \qquad \frac{\Gamma \vdash V_+ : A_+ \ ;\ \Delta}{\Gamma \ ;\ \neg\,[V_+] : \neg A_+ \vdash \Delta}\ \neg L$$

$$\frac{c : (\Gamma \vdash \alpha^+ : A_+, \Delta)}{\Gamma \vdash \mu({\uparrow}[\alpha^+].c) : {\uparrow}A_+ \mid \Delta}\ {\uparrow}R \qquad \frac{\Gamma \mid e_+ : A_+ \vdash \Delta}{\Gamma \ ;\ {\uparrow}[e_+] : {\uparrow}A_+ \vdash \Delta}\ {\uparrow}L$$

FIGURE 4.10. Focused logical typing rules for polarized system L: with positive connectives (0, 1, $\oplus$, $\otimes$, $\sim$, $\downarrow$) and negative connectives ($\top$, $\bot$, $\&$, $\parr$, $\to$, $\neg$, $\uparrow$).

would place a negative term or a positive co-term inside of a construction, breaking the convention.

Next, let's consider the dynamic approach with the extra focusing rewrite rules shown in Figure 4.11. These extra reductions are just enough to prevent the troublesome commands from getting stuck. For example, the $\beta_{\mathcal{P}}^{\oplus}$-stuck command between (co-)terms of type $A_+ \oplus B_+$, $\langle \iota_1 (\mu\alpha^+.c) \| \tilde{\mu}[\iota_1 (x^+).c_1 \mid \iota_2 (y^+).c_2] \rangle$, can now proceed by a $\varsigma_{\mathcal{P}}^{\oplus}$ reduction on the immediate sub-term:

$$\begin{aligned}
&\left\langle \iota_1 \left(\mu\alpha^+.c\right) \middle\| \tilde{\mu}\left[\iota_1 \left(x^+\right).c_1 \mid \iota_2 \left(y^+\right).c_2\right] \right\rangle \\
&\to_{\beta_{\mathcal{P}}^{\oplus}} \left\langle \mu\gamma^+. \left\langle \mu\alpha^+.c \middle\| \tilde{\mu}y^+. \left\langle \iota_1 \left(y^+\right) \middle\| \gamma^+ \right\rangle \right\rangle \middle\| \tilde{\mu}\left[\iota_1 \left(x^+\right).c_1 \mid \iota_2 \left(y^+\right).c_2\right] \right\rangle
\end{aligned}$$

Likewise, the $\beta_{\mathcal{P}}^{\to}$-stuck command between (co-)terms of type $A_+ \to B_-$, $\langle \mu([x^+ \cdot \beta^-].c) \| (\mu\alpha^+.c_1) \cdot [\tilde{\mu}y^-.c_2] \rangle$, can also now proceed by a $\varsigma_{\mathcal{P}}^{\to}$ reduction on the immediate sub-co-term:

$$\begin{aligned}
&\left\langle \mu\left([x^+ \cdot \beta^-].c\right) \middle\| (\mu\alpha^+.c_1) \cdot [\tilde{\mu}y^-.c_2] \right\rangle \\
&\to_{\varsigma_{\mathcal{P}}^{\to}} \left\langle \mu\left([x^+ \cdot \beta^-].c\right) \middle\| \tilde{\mu}x^+. \left\langle \mu\alpha^+.c_1 \middle\| \tilde{\mu}y^+. \left\langle x^+ \middle\| y^+ \cdot [\tilde{\mu}y^-.c_2] \right\rangle \right\rangle \right\rangle
\end{aligned}$$

The combination of $\beta_{\mathcal{P}}$ and $\varsigma_{\mathcal{P}}$ reductions gives us enough tools for a well-behaved extension the core $\mu\tilde{\mu}$ operational semantics. Because $\varsigma_{\mathcal{P}}$ operates on (co-)terms instead of commands, we must extend the set of polarized evaluation contexts D to reduce (co-)terms when necessary as follows:

$$D \in \mathit{EvalCxt}_{\mathcal{P}} ::= \Box \mid \langle \Box \| e_+ \rangle \mid \langle v_- \| \Box \rangle$$

This gives us the $\mu_{\mathcal{P}}\tilde{\mu}_{\mathcal{P}}\beta_{\mathcal{P}}\varsigma_{\mathcal{P}}$ operational semantics ($\mapsto_{\mu_{\mathcal{P}}\tilde{\mu}_{\mathcal{P}}\beta_{\mathcal{P}}\varsigma_{\mathcal{P}}}$), which is strong enough to compute results of the following form:

$$\begin{aligned}
\mathit{FinalCommand}_{\mathcal{P}} &::= \left\langle V_+ \middle\| \alpha^+ \right\rangle \mid \left\langle x^+ \middle\| E_+^s \right\rangle \mid \left\langle x^- \middle\| E_- \right\rangle \mid \left\langle V_-^s \middle\| \alpha^- \right\rangle \\
V_-^s \in \mathit{SimpleValue}_- &= \left\{ v_- \in \mathit{Term}_- \mid v_- \neq_\alpha \mu\alpha^-.c \right\} \\
E_+^s \in \mathit{SimpleCoValue}_+ &= \left\{ e_+ \in \mathit{CoTerm}_+ \mid e_+ \neq_\alpha \tilde{\mu}x^+.c \right\}
\end{aligned}$$

When considering only well-typed commands, we get the standard safety theorem saying that well-typed commands always reduce to a final command shown above similar to the dual calculi.

Positive $\varsigma_{\mathcal{P}}$ rules:

$$\left.\begin{array}{lr}
(\varsigma^{0}_{\mathcal{P}}) & \text{no } \varsigma^{0}_{\mathcal{P}} \text{ rule} \\
(\varsigma^{1}_{\mathcal{P}}) & \text{no } \varsigma^{1}_{\mathcal{P}} \text{ rule} \\
(\varsigma^{\oplus}_{\mathcal{P}}) & \pi_i\left[v_+\right] \succ_{\varsigma^{\oplus}_{\mathcal{P}}} \mu\alpha^+.\left\langle v_+ \middle\| \tilde{\mu}y^+.\left\langle \pi_i\left[y^+\right] \middle\| \alpha^+\right\rangle\right\rangle \\
(\varsigma^{\otimes}_{\mathcal{P}}) & \left(v_+, v'_+\right) \succ_{\varsigma^{\otimes}_{\mathcal{P}}} \mu\alpha^+.\left\langle v_+ \middle\| \tilde{\mu}y^+.\left\langle \left(y^+, v'_+\right) \middle\| \alpha^+\right\rangle\right\rangle \\
(\varsigma^{\otimes}_{\mathcal{P}}) & \left(V_+, v_+\right) \succ_{\varsigma^{\otimes}_{\mathcal{P}}} \mu\alpha^+.\left\langle v_+ \middle\| \tilde{\mu}y^+.\left\langle \left(V_+, y^+\right) \middle\| \alpha^+\right\rangle\right\rangle \\
(\varsigma^{\sim}_{\mathcal{P}}) & \sim(e_-) \succ_{\varsigma^{\sim}_{\mathcal{P}}} \mu\alpha^+.\left\langle \mu\beta^-.\left\langle \sim\left(\beta^-\right) \middle\| \alpha^+\right\rangle \middle\| e_-\right\rangle \\
(\varsigma^{\downarrow}_{\mathcal{P}}) & \text{no } \varsigma^{\downarrow}_{\mathcal{P}} \text{ rule}
\end{array}\right\}
\begin{array}{l}
v_+ \notin Value_{\mathcal{P}}, \\
e_- \notin CoValue_{\mathcal{P}}, \\
\alpha^+, \beta^-, y^+ \text{ fresh}
\end{array}$$

Negative $\varsigma_{\mathcal{P}}$ rules:

$$\left.\begin{array}{lr}
(\varsigma^{\top}_{\mathcal{P}}) & \text{no } \varsigma^{\top}_{\mathcal{P}} \text{ rule} \\
(\varsigma^{\bot}_{\mathcal{P}}) & \text{no } \varsigma^{\bot}_{\mathcal{P}} \text{ rule} \\
(\varsigma^{\&}_{\mathcal{P}}) & \pi_i\left[e_-\right] \succ_{\varsigma^{\&}_{\mathcal{P}}} \tilde{\mu}x^-.\left\langle \mu\beta^-.\left\langle x^- \middle\| \pi_i\left[\beta^-\right]\right\rangle \middle\| e_-\right\rangle \\
(\varsigma^{\parr}_{\mathcal{P}}) & \left[e_-, e'_-\right] \succ_{\varsigma^{\parr}_{\mathcal{P}}} \tilde{\mu}x^+.\left\langle \mu\beta^-.\left\langle x^+ \middle\| \left[\beta^-, e'_-\right]\right\rangle \middle\| e_-\right\rangle \\
(\varsigma^{\parr}_{\mathcal{P}}) & \left[E_-, e_-\right] \succ_{\varsigma^{\parr}_{\mathcal{P}}} \tilde{\mu}x^+.\left\langle \mu\beta^-.\left\langle x^+ \middle\| \left[E_-, \beta^-\right]\right\rangle \middle\| e_-\right\rangle \\
(\varsigma^{\to}_{\mathcal{P}}) & v_+ \cdot e'_- \succ_{\varsigma^{\to}_{\mathcal{P}}} \tilde{\mu}x^+.\left\langle v_+ \middle\| \tilde{\mu}y^+.\left\langle x^+ \middle\| y^+ \cdot e'_-\right\rangle\right\rangle \\
(\varsigma^{\to}_{\mathcal{P}}) & V_+ \cdot e_- \succ_{\varsigma^{\to}_{\mathcal{P}}} \tilde{\mu}x^+.\left\langle \mu\beta^-.\left\langle x^+ \middle\| V_+ \cdot \beta^-\right\rangle \middle\| e_-\right\rangle \\
(\varsigma^{\neg}_{\mathcal{P}}) & \neg\left[v_+\right] \succ_{\varsigma^{\neg}_{\mathcal{P}}} \tilde{\mu}x^+.\left\langle v_+ \middle\| \tilde{\mu}y^+.\left\langle x^+ \middle\| \neg\left[y^+\right]\right\rangle\right\rangle \\
(\varsigma^{\uparrow}_{\mathcal{P}}) & \text{no } \varsigma^{\uparrow}_{\mathcal{P}} \text{ rule}
\end{array}\right\}
\begin{array}{l}
e_- \notin CoValue_{\mathcal{P}}, \\
v_+ \notin Value_{\mathcal{P}} \\
x^-, y^+, \beta^- \text{ fresh}
\end{array}$$

FIGURE 4.11. The focusing ς laws for polarized system L: with two unit types (1, $\top$), two empty types (0, $\bot$), (co-)products ($\&$, $\oplus$), (co-)pairs ($\otimes$, $\parr$), two negations ($\sim$, $\neg$), functions ($\to$), and two polarity shifts ($\downarrow$, $\uparrow$).

Theorem 4.1 (Progress and preservation)**.** *For any system L command* $c : (\Gamma \vdash \Delta)$*:*

a) Progress: *c is a polarized final command or there is a command c' such that $c \mapsto_{\mu_{\mathcal{P}}\tilde{\mu}_{\mathcal{P}}\beta_{\mathcal{P}}\varsigma_{\mathcal{P}}} c'$, and*

b) Preservation: *if $c \mapsto_{\mu_{\mathcal{P}}\tilde{\mu}_{\mathcal{P}}\beta_{\mathcal{P}}\varsigma_{\mathcal{P}}} c'$, then $c' : (\Gamma \vdash \Delta)$.*

Proof. The proof is analogous to the proof of Theorem 3.3. Progress follows by induction on the typing derivation of $c : (\Gamma \vdash \Delta)$, which is assured because

- every v_+ is either a value, an output abstraction, or a $\varsigma_{\mathcal{P}}$ redex,
- every e_+ is either an input abstraction or in $\mathit{SimpleCoValue}_+$,
- every v_- is either an output abstraction or in $\mathit{SimpleValue}_-$, and
- every e_- is either a co-value, an input abstraction, or a $\varsigma_{\mathcal{P}}$ redex.

Therefore, if the cut is neither final nor reducible, then either its positive term or negative co-term $\varsigma_{\mathcal{P}}$-reduces. Preservation follows by cases on all the possible rewriting rules using the substitution principle for typing derivations similar to Theorem 3.3, so that

- if $c \succ_{\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}\beta\varsigma} c'$ then $c : (\Gamma \vdash \Delta)$ implies $c' : (\Gamma \vdash \Delta)$,
- if $v \succ_{\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}\beta\varsigma} v'$ then $v : (\Gamma \vdash \Delta)\, C$ implies $v' : (\Gamma \vdash \Delta)\, C$, and
- if $e \succ_{\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}\beta\varsigma} e'$ then $e : (\Gamma \vdash \Delta)\, C$ implies $e' : (\Gamma \vdash \Delta)\, C$. □

Also, much like the dual calculi, the two methods of focusing correspond to one another, applying the same essential transformations either during execution or as a pre-processing pass. More specifically, the focused sub-syntax of polarized system L contains exactly the $\varsigma_{\mathcal{P}}$-normal forms of system L, and therefore every command, term, and co-term can be $\varsigma_{\mathcal{P}}$ reduced into the focused sub-syntax.

Theorem 4.2 (Focusing)**.** *Every polarized system L command, term, and co-term is in the focused sub-syntax if and only if it is a $\varsigma_{\mathcal{P}}$-normal form. Furthermore, for every polarized system L command c, term v, and co-term e, there is a focused command c', term v', and co-term e' such that $c \twoheadrightarrow_{\varsigma_{\mathcal{P}}} c'$, $v \twoheadrightarrow_{\varsigma_{\mathcal{P}}} v'$, and $e \twoheadrightarrow_{\varsigma_{\mathcal{P}}} e'$.*

Proof. First, the fact that the command and (co-)term in the focused sub-syntax are in one-for-one correspondence with $\varsigma_{\mathcal{P}}$-normal forms follows by induction on the syntax of polarized system L commands and (co-)terms.

Second, observe that the $\varsigma_{\mathcal{P}}$ reduction theory is strongly normalizing because each reduction reduces the number of non-(co-)values within term and co-term constructions which serves as a normalization measure. Therefore, every command and (co-)term has a unique $\varsigma_{\mathcal{P}}$-normal form, which by the first point must lie within the focused sub-syntax of polarized system L. □

Self-Duality

System L exhibits a logical duality similar to the dual calculi (see Section 3.3). However, the dual calculi are two separate dual calculi—one call-by-value and one call-by-name—that share common syntax and types, polarized system L is *self-dual*. In other words, we can say that polarized system L internalizes the notion of duality inside of itself, so that it gives a single, complete, and self-contained language for discussing and using dual concepts. This is because polarization lets us incorporate *both* call-by-name and call-by-value constructs and evaluation. In the dual calculi, call-by-value programs are dualized into call-by-name ones, and vice versa, which lie in the two separate interpretations of the same syntax. But polarized system L contains both call-by-value and call-by-name fragments, so that there is no need for a separate calculus and a change of interpretation to accomodate the inversion of control flow caused by duality.

As with the dual calculi, the self-duality of polarized system L resembles the de Morgan laws, where truth is dual to falsehood and conjunction is dual to conjunction. However, polarity explicitly reveals another aspect of duality that was implicit in the dual calculus: duality also reverses the polarity of types and programs. So 0 is dual to $\top$, 1 is dual to $\bot$, $\oplus$ is dual to $\&$, and $\otimes$ is dual to $⅋$. The polarity reversal corresponds to the fact that the dynamic semantics of call-by-value is dual to that of call-by-name. This also means that, whereas the single negation connective was self-dual in the dual calculi, the two polarities of negation ($\sim$ and $\neg$) are dual to one another. Likewise, the two polarity shifts ($\downarrow$ and $\uparrow$) are also dual connectives.

The only lack of symmetry is with function types $A_+ \to B_-$ which lack their dual counterpart, as was the case in both LK (Section 3.1) and the dual calculi (Section 3.3). As is now the standard procedure, this asymmetry is easily remedied

$$
\begin{array}{ll}
v_+ \in \mathit{Term}_+ ::= \ldots \mid e_- \cdot v_+ & e_+ \in \mathit{CoTerm}_+ ::= \ldots \mid \tilde{\mu}\left[(\alpha^- \cdot y^+).c\right] \\
V_+ \in \mathit{Value}_+ ::= \ldots \mid E_- \cdot V_+ & A_+, B_+, C_+ \in \mathit{Type}_+ ::= \ldots \mid B_+ - A_-
\end{array}
$$

$$
\frac{\Gamma' \mid e_- : A_- \vdash \Delta' \quad \Gamma \vdash v_+ : B_+ \mid \Delta}{\Gamma, \Gamma' \vdash e_- \cdot v_+ : B_+ - A_- \mid \Delta, \Delta'} -R
\qquad
\frac{c : (\Gamma, y^+ : B_+ \vdash \alpha^- : A_- \Delta)}{\Gamma \mid \tilde{\mu}[(\alpha^- \cdot y^+).c] : B_+ - A_- \vdash \Delta} -L
$$

$$
\frac{\Gamma' \,;\, E_- : A_- \vdash \Delta' \quad \Gamma \vdash V_+ : B_+ \,;\, \Delta}{\Gamma, \Gamma' \vdash E_- \cdot V_+ : B_+ - A_- \,;\, \Delta, \Delta'} -R
$$

$$
\begin{array}{lrl}
(\beta^-_{\mathcal{P}}) & \left\langle E_- \cdot V_+ \middle\| \tilde{\mu}\left[(\alpha^- \cdot y^+).c\right]\right\rangle \succ_{\beta^-_{\mathcal{P}}} & c\left\{V_+/y^+, \alpha^- E_-\right\} \\
(\varsigma^-_{\mathcal{P}}) & e_- \cdot v_+ \succ_{\varsigma^-_{\mathcal{P}}} & \mu\alpha^+.\left\langle \mu\beta^+.\left\langle \beta^- \cdot v_+ \middle\| \alpha^+ \right\rangle \middle\| e_- \right\rangle \\
 & & (e \notin \mathit{CoValue}_-, \quad \alpha^+, \beta^- \notin FV(e_- \cdot v_+)) \\
(\varsigma^-_{\mathcal{P}}) & E_- \cdot v_+ \succ_{\varsigma^-_{\mathcal{P}}} & \mu\alpha^+.\left\langle v_+ \middle\| \tilde{\mu}y^+.\left\langle E_- \cdot y^+ \middle\| \alpha^+ \right\rangle\right\rangle \\
 & & (v \notin \mathit{Value}_+, \quad \alpha^+, y^+ \notin FV(E_- \cdot v_+)) \\
(\eta^-_{\mathcal{P}}) & e_+ : B_+ - A_- \prec_{\eta^-_{\mathcal{P}}} & \tilde{\mu}\left[(\alpha^- \cdot y^+).\left\langle \alpha^- \cdot y^+ \middle\| e \right\rangle\right]
\end{array}
$$

FIGURE 4.12. Extending polarized system L with subtraction (−), the dual of implication (→).

by adding subtraction types $B_+ - A_-$ as the dual counterpart to function types as shown in Figure 4.12. Syntactically, this presentation of subtraction is the same as in the dual calculi, except that we use a case abstraction co-term $\tilde{\mu}[(\alpha \cdot y).c]$ to match the system L style of pattern-matching function terms instead of a λ abstracting a co-variable over a co-term. Additionally, the fact that the function type $A_+ \to B_-$ mixes the two polarities is reflected in the subtraction type $B_+ - A_-$. With symmetry restored, we formally define the duality of polarized system L types in Figure 4.13 and programs in Figure 4.14.

The self-duality of polarized system L exhibits the same pleasant properties as duality of the dual calculi from Section 3.3: the duality relation is involutive, respects static semantics (typing), and respects dynamic semantics (reduction). The major departure from the dual calculi is that all the dynamic semantics and rewriting rules are contained within the same polarized language, instead of being split between two interpretations of the same syntax.

$$
\begin{aligned}
(X^+)^\perp &\triangleq \overline{X}^- & (X^-)^\perp &\triangleq \overline{X}^+ \\
0^\perp &\triangleq \top & \top^\perp &\triangleq 0 \\
1^\perp &\triangleq \bot & \bot^\perp &\triangleq 1 \\
(A_+ \oplus B_+)^\perp &\triangleq (A_+^\perp) \mathbin{\&} (B_+^\perp) & (A_- \mathbin{\&} B_-)^\perp &\triangleq (A_-^\perp) \oplus (B_-^\perp) \\
(A_+ \otimes B_+)^\perp &\triangleq (A_+^\perp) \parr (B_+^\perp) & (A_- \parr B_-)^\perp &\triangleq (A_-^\perp) \otimes (B_-^\perp) \\
(B_+ - A_-)^\perp &\triangleq (A_-^\perp) \to (B_+^\perp) & (A_+ \to B_-)^\perp &\triangleq (B_-^\perp) - (A_+^\perp) \\
({\sim} A_-)^\perp &\triangleq \neg(A_-^\perp) & (\neg A_+)^\perp &\triangleq {\sim}(A_+^\perp) \\
({\downarrow} A_-)^\perp &\triangleq {\uparrow}(A_-^\perp) & ({\uparrow} A_+)^\perp &\triangleq {\downarrow}(A_+^\perp)
\end{aligned}
$$

FIGURE 4.13. The self-duality of system L types: with two unit types $(1, \top)$, two empty types $(0, \bot)$, (co-)products $(\oplus, \&)$, (co-)pairs $(\otimes, \parr)$, (co-)functions $(\to, -)$, and negations $(\sim, \neg)$, and two polarity shifts $(\downarrow, \uparrow)$.

Theorem 4.3 (Involutive duality)**.** *The duality operation* $\_^\perp$ *on environments, sequents, types, commands, terms, and co-terms is involutive, so that* $\_^{\perp\perp}$ *is the identity transformation.*

Proof. By induction on the definition of the duality operation $\_^\perp$, similar to the proof of Theorem 3.6. □

Theorem 4.4 (Static duality)**.**

a) $c : (\Gamma \vdash \Delta)$ *is well-typed if and only if* $c^\perp : (\Delta^\perp \vdash \Gamma^\perp)$ *is.*

b) $\Gamma \vdash v : A \mid \Delta$ *is well-typed if and only if* $\Delta^\perp \mid v^\perp : A^\perp \vdash \Gamma^\perp$ *is.*

c) $\Gamma \mid e : A \vdash \Delta$ *is well-typed if and only if* $\Delta^\perp \vdash e^\perp : A^\perp \mid \Gamma^\perp$ *is.*

Furthermore, if a command, term, or co-term lies in the focused sub-syntax, then so does its dual.

Proof. By induction on the typing derivation, similar to the proof of Theorem 3.7. □

Theorem 4.5 (Dynamic duality)**.** *a)* $c \succ_{\mu_{\mathcal{P}}\tilde{\mu}_{\mathcal{P}}\beta_{\mathcal{P}}} c'$ *if and only if* $c^\perp \succ_{\mu_{\mathcal{P}}\tilde{\mu}_{\mathcal{P}}\beta_{\mathcal{P}}} c'^\perp$.

b) $v \succ_{\eta_\mu \varsigma_{\mathcal{P}}} v'$ *if and only if* $v^\perp \succ_{\eta_{\tilde{\mu}} \varsigma_{\mathcal{P}}} v'^\perp$.

c) $e \succ_{\eta_{\tilde{\mu}} \varsigma_{\mathcal{P}}} e'$ *if and only if* $e^\perp \succ_{\eta_\mu \varsigma_{\mathcal{P}}} e'^\perp$.

Proof. By cases on the respective rewriting rules using the fact that substitution commutes with duality, similar to the proof of Theorem 3.8. □

$$\langle v_+ \| e_+ \rangle^\perp \triangleq \left\langle e_+^\perp \middle\| v_+^\perp \right\rangle \qquad \langle v_- \| e_- \rangle^\perp \triangleq \left\langle e_-^\perp \middle\| v_-^\perp \right\rangle$$

$$\begin{aligned}
(x^+)^\perp &\triangleq \overline{x}^- \\
(\mu\alpha^+.c)^\perp &\triangleq \tilde{\mu}\overline{\alpha}^-.c^\perp \\
()^\perp &\triangleq [] \\
\iota_1\,(v_+)^\perp &\triangleq \pi_1\left[v_+^\perp\right] \\
\iota_2\,(v_+)^\perp &\triangleq \pi_2\left[v_+^\perp\right] \\
\left(v_+, v'_+\right)^\perp &\triangleq \left[v_+^\perp, v'^\perp_+\right] \\
(e_- \cdot v_+)^\perp &\triangleq e_-^\perp \cdot v_+^\perp \\
\sim(e_-)^\perp &\triangleq \neg\left[e_-^\perp\right] \\
{\downarrow}(v_-)^\perp &\triangleq {\uparrow}\left[v_-^\perp\right]
\end{aligned}$$

$$\begin{aligned}
[\alpha^+]^\perp &\triangleq \overline{\alpha}^- \\
[\tilde{\mu}x^+.c]^\perp &\triangleq \mu\overline{x}^-.c^\perp \\
\tilde{\mu}[]^\perp &\triangleq \mu() \\
\tilde{\mu}[().c]^\perp &\triangleq \mu\Big([].c^\perp\Big) \\
\tilde{\mu}\Big[\iota_1\left(x^+\right).c \mid \iota_2\left(y^+\right).c'\Big]^\perp &\triangleq \mu\Big(\pi_1\left[\overline{x}^-\right].c^\perp \mid \pi_2\left[\overline{y}^-\right].c'^\perp\Big) \\
\tilde{\mu}\Big[\left(x^+, y^+\right).c\Big]^\perp &\triangleq \mu\Big(\left[\overline{x}^-, \overline{y}^-\right].c^\perp\Big) \\
\tilde{\mu}\Big[(\alpha^- \cdot x^+).c\Big]^\perp &\triangleq \mu\Big([\overline{\alpha}^+ \cdot \overline{x}^-].c^\perp\Big) \\
\tilde{\mu}\Big[\sim\left(\alpha^-\right).c\Big]^\perp &\triangleq \mu\Big(\neg\left[\overline{\alpha}^+\right].c^\perp\Big) \\
\tilde{\mu}\Big[{\downarrow}\left(x^-\right).c\Big]^\perp &\triangleq \mu\Big({\uparrow}\left[\overline{x}^+\right].c^\perp\Big)
\end{aligned}$$

$$\begin{aligned}
(x^-)^\perp &\triangleq \overline{x}^+ \\
(\mu\alpha^-.c)^\perp &\triangleq \tilde{\mu}\overline{\alpha}^+.c^\perp \\
\mu()^\perp &\triangleq \tilde{\mu}[] \\
\mu([].c)^\perp &\triangleq \tilde{\mu}\Big[().c^\perp\Big] \\
\mu\Big(\pi_1\left[\alpha^-\right].c \mid \pi_2\left[\beta^-\right].c'\Big)^\perp &\triangleq \tilde{\mu}\Big[\iota_1\left(\overline{\alpha}^+\right).c^\perp \mid \iota_2\left(\overline{\beta}^+\right).c'^\perp\Big] \\
\mu\Big(\left[\alpha^-, \beta^-\right].c\Big)^\perp &\triangleq \tilde{\mu}\Big[\left(\overline{\alpha}^+, \overline{\beta}^+\right).c^\perp\Big] \\
\mu\Big([x^+ \cdot \alpha^-].c\Big)^\perp &\triangleq \tilde{\mu}\Big[(\overline{x}^- \cdot \overline{\alpha}^+).c^\perp\Big] \\
\mu\Big(\neg\left[x^+\right].c\Big)^\perp &\triangleq \tilde{\mu}\Big[\sim\left(\overline{x}^-\right).c^\perp\Big] \\
\mu\Big({\uparrow}\left[\alpha^+\right].c\Big)^\perp &\triangleq \tilde{\mu}\Big[{\downarrow}\left(\overline{\alpha}^-\right).c^\perp\Big]
\end{aligned}$$

$$\begin{aligned}
[\alpha^-]^\perp &\triangleq \overline{\alpha}^+ \\
[\tilde{\mu}x^-.c]^\perp &\triangleq \mu\overline{x}^+.c^\perp \\
[]^\perp &\triangleq () \\
\pi_1\,[e_-]^\perp &\triangleq \iota_1\left(e_-^\perp\right) \\
\pi_2\,[e_-]^\perp &\triangleq \iota_2\left(e_-^\perp\right) \\
\left[e_-, e'_-\right]^\perp &\triangleq \left(e_-^\perp, e'^\perp_-\right) \\
[v_+ \cdot e_-]^\perp &\triangleq v_+^\perp \cdot e_-^\perp \\
\neg\,[v_+]^\perp &\triangleq \sim\left(v_+^\perp\right) \\
{\uparrow}[e_+]^\perp &\triangleq {\downarrow}\left(e_+^\perp\right)
\end{aligned}$$

FIGURE 4.14. The self-duality of system L programs: with two unit types $(1, \top)$, two empty types $(0, \bot)$, (co-)products $(\oplus, \&)$, (co-)pairs $(\otimes, ⅋)$, (co-)functions $(\to, -)$, and negations $(\sim, \neg)$, and two polarity shifts $(\downarrow, \uparrow)$.

CHAPTER V

Data and Co-Data

The ramifications of treating the sequent calculus as a programming language (Curien & Herbelin, 2000; Wadler, 2003; Zeilberger, 2008b; Munch-Maccagnoni, 2009) have elucidated issues that arise in programs, including the interplay between strict and lazy evaluation in programs and types. When interpreted as a computational framework, the sequent calculus reveals a diversity of connectives that is easy to overlook in the tradition of the λ-calculus. However, this diversity can become overwhelming. We now have several connectives for representing similar logical ideas: two connectives each for conjunction, disjunction, negation, and so on.

Additionally, there are still some questions that have not been addressed. For instance, how do other evaluation strategies, like call-by-need (Ariola *et al.*, 1995; Ariola & Felleisen, 1997; Maraist *et al.*, 1998),[1] fit into the picture? If we follow the story of polarized logic, that the polarity determines evaluation order, then there is no room—by definition there are only two polarities so we can only directly account for two evaluation strategies with this approach.

We now aim to tame the abundance of connectives found in the sequent calculus. Can we find a single pattern that encompasses every single connective we have discussed so far in the sequent calculus? That way, rather than cataloguing the many different connectives on a case-by-case basis, we can direct our attention on the commonalities underlying them all. As a tool for analysis, we summarize a broad family of types occurring in the sequent calculus, whose static and dynamic properties all derive from a small core. As a tool for synthesis, we use the patterns underpinning the connectives as a mechanism facilitating the exploration of new connectives.

Furthermore, we look for a more general classification of evaluation strategies, in an effort to capture the essence of strategies, that goes beyond the duality between call-by-value and call-by-name evaluation. In order to account for other

This chapter is a new text based on the ideas and results from (Downen & Ariola, 2014c).

[1] Call-by-need can be thought of as a memoizing version of call-by-name where the arguments to function calls are evaluated on demand, like in call-by-name, but where the value of an argument is remembered so that it is computed only once, like in call-by-value.

evaluation strategies like call-by-need, we need to step outside of the polarization hypothesis, which assumed that every evaluation strategy corresponds to one of the (two!) polarities. Instead, we look at a treatment of strategy based on its impact on substitution. The substitution-based characterization of evaluation strategies is general enough to describe call-by-need evaluation and also generalizes polarization as a mechanism for combining different evaluation strategies within a single program.

Our approach to understanding the dynamic behavior of the various connectives is the same as the traditional approach from the λ-calculus: the dynamic meaning of all connectives are characterized by β and η laws. We will first investigate these principles as symmetric equations, rather than non-symmetric reductions, which lets us understand β and η laws that are valid for *any* evaluation strategy. Besides maintaining similarity with the simply typed λ-calculus, the equational theory avoids the conflict between extensionality and control that arises in rewriting theories for classical logic (David & Py, 2001). Instead, we drive the (untyped) reduction theory and operational semantics for all the connectives, which includes the operational β and focusing ς rewriting rules previously studied in Chapters III and IV, is justified in terms of the fundamental β and η equations.

The Essence of Evaluation: Substitutability

As we have seen previously in Chapters III and IV, there are many different languages for the sequent calculus (Curien & Herbelin, 2000; Wadler, 2003; Herbelin, 2005; Munch-Maccagnoni, 2009; Munch-Maccagnoni & Scherer, 2015) that are all based on the same structural core $\mu\tilde{\mu}$-calculus that was explored in Section 3.2. This core, as was in Figure 3.7, forms the basis of naming in the sequent calculus via variables and co-variables as well as input and output abstractions. Further still, the fundamental dilemma of computation in classical sequent calculus lies wholly within this core. The root cause of non-determinism, non-confluence, and incoherence is a conflict between the input and output abstractions, where each one tries to take control over the future path of evaluation. Therefore, before we tackle evaluation of the language with (co-)data types, we will first focus on how to characterize the resolutions to the fundamental dilemma in the structural core of the sequent calculus.

$$
\begin{array}{lll}
(\mu_{\mathcal{S}}) & \langle \mu\alpha.c \| E\rangle \succ_{\mu_{\mathcal{S}}} c\,\{E/\alpha\} & (E \in \mathit{CoValues}_{\mathcal{S}}) \\
(\tilde{\mu}_{\mathcal{S}}) & \langle V \| \tilde{\mu}x.c\rangle \succ_{\tilde{\mu}_{\mathcal{S}}} c\,\{V/x\} & (V \in \mathit{Values}_{\mathcal{S}}) \\
(\eta_{\mu}) & \mu\alpha.\,\langle v \| \alpha\rangle \succ_{\eta_{\mu}} v & (\alpha \notin FV(v)) \\
(\eta_{\tilde{\mu}}) & \tilde{\mu}x.\,\langle x \| e\rangle \succ_{\eta_{\tilde{\mu}}} e & (x \notin FV(e))
\end{array}
$$

FIGURE 5.1. A parametric theory, $\mu\tilde{\mu}_{\mathcal{S}}$, for the core $\mu\tilde{\mu}$-calculus.

Recall that the source of the conflict in the structural core of the sequent calculus comes from the two opposing rules for implementing substitution:

$$\langle \mu\alpha.c \| e\rangle \succ_{\mu} c\,\{e/\alpha\} \qquad\qquad \langle v \| \tilde{\mu}x.c\rangle \succ_{\tilde{\mu}} c\,\{v/x\}$$

As stated, a command like $\langle \mu\_.c_1 \| \tilde{\mu}\_.c_2\rangle$, where the (co-)variables are never used, is equal to both c_1 and c_2, so any two arbitrary commands may be considered equal. The language-based solution to this dilemma from Chapter III is to restrict one of the two rules to remove the conflict—the μ rule is restricted to co-values to implement a form of call-by-name evaluation or the $\tilde{\mu}$ rule is restricted to values implement a form of call-by-value evaluation. However, in lieu of inventing various different languages with different evaluation strategies for mitigating the conflict, let's instead admit restrictions on *both* directions of substitution to values and co-values:

$$\langle \mu\alpha.c \| E\rangle \succ_{\mu_{\mathcal{S}}} c\,\{E/\alpha\} \qquad\qquad \langle V \| \tilde{\mu}x.c\rangle \succ_{\tilde{\mu}_{\mathcal{S}}} c\,\{V/x\}$$

while leaving the specifics of what constitutes a substitutable value V and a substitutable co-value E open-ended. That is to say, we make the sets of values ($V \in \mathit{Values}_{\mathcal{S}}$) and co-values ($E \in \mathit{CoValues}_{\mathcal{S}}$) a *parameter* of the theory, in the same sense as Ronchi Della Rocca & Paolini's (2004) parametric λ-calculus, that may be filled in at a later time. A choice of a specific value set $\mathit{Values}_{\mathcal{S}}$ and co-value set $\mathit{CoValues}_{\mathcal{S}}$ makes up a *substitution strategy* $\mathcal{S} = (\mathit{Values}_{\mathcal{S}}, \mathit{CoValues}_{\mathcal{S}})$. The full parametric equational theory $\mu\tilde{\mu}$ for the structural core (Downen & Ariola, 2014c) is given in Figure 5.1, where we denote a particular instance of for a chosen substitution strategy $\mathcal{S}$ as $\mu\tilde{\mu}_{\mathcal{S}}$. Since the rules for extensionality of input and output abstractions did not cause any issue, we leave them alone.

By leaving the choice of dual substitution restrictions open as parameters, the *same* parametric theory may describe the semantics different evaluation strategies by instantiating the parameters in different ways. As per Remark 2.3, we can derive reduction and equational theories from the $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}$ rewriting rules from Figure 5.1 as their compatible-reflexive-transitive and compatible-reflexive-symmetric-transitive closures, respectively. So, given a particular substitution strategy $\mathcal{S}$, the $\mathcal{S}$ instance of the parametric reduction and equational theories, denoted $\mu\tilde{\mu}_{\mathcal{S}}$, is obtained by instantiating the set of values and co-values with $\mathcal{S}$. The one constraint on the substitution strategy is that we always assume that variables are values, and co-variables are co-values, since our restriction on the μ and $\tilde{\mu}$ axioms mean that they can only ever stand in for unknown value and co-values.

If we want to characterize an operational semantics as well, we also need to specify the evaluation contexts in which the standard reduction may occur. Therefore, we say that an *evaluation strategy* $\mathcal{S}$ (or just *strategy* for short) is a substitution strategy together with a set of evaluation contexts ($D \in \mathit{EvalCxt}_{\mathcal{S}}$) that yield a command when filled with a command, term, or co-term as appropriate, and which includes at least the following contexts:

- $\Box \in \mathit{EvalCxt}_{\mathcal{S}}$,
- $\langle \Box \| E \rangle \in \mathit{EvalCxt}_{\mathcal{S}}$ for all $E \in \mathit{CoValue}_{\mathcal{S}}$, and
- $\langle V \| \Box \rangle \in \mathit{EvalCxt}_{\mathcal{S}}$ for all $V \in \mathit{Value}_{\mathcal{S}}$.

So a choice of evaluation strategy $\mathcal{S}$ gives us the $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}$ operational semantics that is closed under $\mathit{EvalCxt}_{\mathcal{S}}$ contexts.

The previous characterizations of call-by-value and call-by-name from Chapter III come out as particular instances of the parametric theory. For example, we can define the call-by-value strategy $\mathcal{V}$, shown in Figure 5.2, by restricting the set of values to exclude output abstractions, leaving variables as the only value, and letting every co-term be a co-value. In effect, this decision restricts the $\tilde{\mu}$ rule in the usual way for call-by-value while letting the μ rule be unrestricted. In addition, the $\mathcal{V}$ evaluation contexts only permit reduction at the top of a command or one of its immediate sub-(co-)terms, favoring the term side over the co-term side. The call-by-name strategy $\mathcal{N}$ is defined in the dual way by letting every term be a value and restricting the set of co-values to exclude input abstractions, leaving co-variables as the only co-value.

$$\begin{array}{rclcrcl}
V \in \mathit{Value}_{\mathcal{V}} & ::= & x & \qquad & V \in \mathit{Value}_{\mathcal{N}} & ::= & v \\
E \in \mathit{CoValue}_{\mathcal{V}} & ::= & e & & E \in \mathit{CoValue}_{\mathcal{N}} & ::= & \alpha \\
D \in \mathit{EvalCxt}_{\mathcal{V}} & ::= & \Box \mid \langle \Box \| e \rangle \mid \langle V \| \Box \rangle & & D \in \mathit{EvalCxt}_{\mathcal{N}} & ::= & \Box \mid \langle v \| \Box \rangle \mid \langle \Box \| E \rangle
\end{array}$$

FIGURE 5.2. Call-by-value ($\mathcal{V}$) and call-by-name ($\mathcal{N}$) strategies for the core $\mu\tilde{\mu}$-calculus.

Again, this choice of values and co-values describes the call-by-name restriction on the μ rule while leaving the $\tilde{\mu}$ rule unrestricted. The $\mathcal{N}$ evaluation contexts also only permit reduction at the top of commands or the immediate sub-(co-)terms, but instead favor the co-term side over the term size.

We can also explore other choices for the parameters that describe strategies besides just call-by-value and call-by-name. For instance, we can characterize a notion of call-by-need in terms of a "lazy call-by-value" strategy $\mathcal{LV}$ shown in Figure 5.3, which characterizes evaluation similar to a previous call-by-need theory for the sequent calculus (Ariola *et al.*, 2011). The intuition for $\mathcal{LV}$ is similar to the call-by-need λ-calculus (Ariola *et al.*, 1995): a non-value term bound to a variable represents a delayed computation that will only be evaluated when it is needed. Then, only once the term has been reduced to a value (in the sense of call-by-value), may it be substituted for the variable. In this way, $\mathcal{LV}$ only performs $\mathcal{V}$ substitutions (which we can see from the fact that the $\mathcal{LV}$ (co-)values are a subset of $\mathcal{V}$ (co-)values), but in a lazy, pull-driven fashion that gives initial priority to the consumer as in $\mathcal{N}$. Therefore, in the command $\langle v_1 \| \tilde{\mu}x.\, \langle v_2 \| \tilde{\mu}y.c \rangle \rangle$, we temporarily ignore v_1 and v_2 and work inside c since this command decomposes into the evaluation context $\langle v_1 \| \tilde{\mu}x.\, \langle v_2 \| \tilde{\mu}y.\Box \rangle \rangle$ surrounding c. If it turns out that c evaluates to $D[\langle x \| E \rangle]$, we are left in the state $\langle v_1 \| \tilde{\mu}x.\, \langle v_2 \| \tilde{\mu}y.D[\langle x \| E \rangle] \rangle \rangle$, where E is a co-value that wants to know something about x, making $\tilde{\mu}x.\, \langle v_2 \| \tilde{\mu}y.D[\langle x \| E \rangle] \rangle$ into a co-value as well. Therefore, if v_1 is a non-value output abstraction, it may take over via the $\mu_{\mathcal{LV}}$ rule, and thus begin evaluation of the value of the demanded variable x.

Due to the symmetry of the sequent calculus, it is straightforward to generate the dual to the call-by-need strategy, which is the "lazy call-by-name" strategy $\mathcal{LN}$ shown in Figure 5.4. This strategy performs a subset of $\mathcal{N}$ substitutions (since $\mathcal{LN}$ (co-)values are a subset of $\mathcal{N}$ (co-)values), but still gives initial priority to the producer as in $\mathcal{V}$. For example, in the command $\langle \mu\alpha.\, \langle \mu\beta.c \| e_2 \rangle \| e_1 \rangle$, we temporarily ignore e_1

$$\begin{aligned} V \in \mathit{Value}_{\mathcal{LV}} &::= x \\ E \in \mathit{CoValue}_{\mathcal{LV}} &::= \alpha \mid \tilde{\mu}x.D[\langle x \| E\rangle] \\ D \in \mathit{EvalCxt}_{\mathcal{LV}} &::= \Box \mid \langle v \| \tilde{\mu}y.D\rangle \mid \langle v \| \Box\rangle \mid \langle \Box \| E\rangle \end{aligned}$$

FIGURE 5.3. "Lazy-call-by-value" ($\mathcal{LV}$) strategy for the core $\mu\tilde{\mu}$-calculus.

$$\begin{aligned} V \in \mathit{Value}_{\mathcal{LN}} &::= x \mid \mu\alpha.D[\langle V \| \alpha\rangle] \\ E \in \mathit{CoValue}_{\mathcal{LN}} &::= \alpha \\ D \in \mathit{EvalCxt}_{\mathcal{LN}} &::= \Box \mid \langle \mu\alpha.D \| e\rangle \mid \langle \Box \| e\rangle \mid \langle V \| \Box\rangle \end{aligned}$$

FIGURE 5.4. "Lazy-call-by-name" ($\mathcal{LN}$) strategy for the core $\mu\tilde{\mu}$-calculus.

and e_2 and work inside c since this command decomposes into the $\mathcal{LN}$ evaluation context $\langle \mu\alpha. \langle \mu\beta.\Box \| e_2\rangle \| e_1\rangle$ surrounding c. If it turns out that c evaluates to $D[\langle V \| \alpha\rangle]$, we are left in the state $\langle \mu\alpha. \langle \mu\beta.D[\langle V \| \alpha\rangle] \| e_2\rangle \| e_1\rangle$, where V is a value that wants to yield a result to α, making $\mu\alpha. \langle \mu\beta.D[\langle V \| \alpha\rangle] \| e_2\rangle$ a value as well. Therefore, if e_1 is a non-co-value input abstraction, it may take over via the $\tilde{\mu}_{\mathcal{LN}}$ rule, and thus begin evaluation of the observation for the demanded co-variable α.

Remark 5.1. Note that, while our primary interest in strategies is to achieve a coherent, confluent theory of deterministic evaluation by avoiding the fundamental dilemma of classical computation, individual strategies are not required to do so. That is to say, it can be meaningful to talk about strategies that yield incoherent theories for the sequent calculus, if we are not interested in properties like confluence. For example, the simplest such strategy is the "unrestricted" strategy, $\mathcal{U}$, for unconstrained and non-deterministic evaluation, which considers every term to be a value and every co-term to be a co-value as shown in Figure 5.5. The $\tilde{\mu}_{\mathcal{U}}\tilde{\mu}_{\mathcal{U}}$ theory effectively ignores the concept of values and co-values, choosing to restrict neither the μ nor $\tilde{\mu}$ rules for substitution, and thereby giving a theory corresponding to Barbanera & Berardi's

$$V \in Value_{\mathcal{U}} ::= v \qquad E \in CoValue_{\mathcal{U}} ::= e \qquad D \in EvalCxt_{\mathcal{U}} ::= \Box$$

FIGURE 5.5. Nondeterministic ($\mathcal{U}$) strategy for the core $\mu\tilde{\mu}$-calculus.

(1994) symmetric λ-calculus for a classical logic that does not consider a restricted evaluation strategy. *End remark* 5.1.

Remark 5.2. Another way to think about substitution strategies, and the parameterized notions of values and co-values, is to consider the essential parts of an equational theory. Typically, equational theories are expressed by a set of axioms (primitive equalities assumed to hold) along with some basic properties or rules for forming larger equations like compatibility reflexivity, symmetry, and transitivity previously discussed in Remark 2.3.

In a language with an internal notion of variables, like the λ-calculus or the core $\mu\tilde{\mu}$-calculus, we also generally expect the equational theory to be closed under substitution. That is to say, if two things are equal, then they should still be equal after substituting the same term for the same variable in both of them. However, this principle often does not always hold in full generality for programming languages. For example, the ML terms $\mathbf{let}\, y = x \,\mathbf{in}\, 5$ and 5 are equal—they will always behave the same in any context. However, if we substitute the term $(\mathtt{print}\ ''\mathtt{hi}''; 1)$ for x in both, we end up with $\mathbf{let}\, y = (\mathtt{print}\ ''\mathtt{hi}''; 1) \,\mathbf{in}\, 5$ and 5, which are no longer equal because one produces an observable side effect (printing the string "hi") and the other does not. Instead, ML supports a restricted substitution principle: if two terms are equal, then they are still equal when we substitute the same *value* (an integer, a pair of values, a function abstraction, ...) for the same variable in both of them. This restriction deftly avoids these kinds of counter-examples.

The exact same issue arises in the classical sequent calculus, since it also includes effects that allow manipulation of control flow. Therefore, we need to restrict the substitution principle in the sequent calculus to only allow substituting values for variables. Additionally, since we have a second form of substitution, we also have a restriction that only allows substituting co-values for co-variables. This leads us to substitution principles that say if two commands (or terms or co-terms) are equal, they must still be equal after substituting *(co-)values* for (co-)variables:

$$\frac{c = c' \quad V \in \mathit{Value}_{\mathcal{S}}}{c\,\{V/x\} = c'\,\{V/x\}}\ \mathit{subst}_{\mathcal{S}} \qquad\qquad \frac{c = c' \quad E \in \mathit{CoValue}_{\mathcal{S}}}{c\,\{E/\alpha\} = c'\,\{E/\alpha\}}\ \mathit{subst}_{\mathcal{S}}$$

and similarly for substitutions in terms and co-terms.

In lieu of the presentation in Figure 5.1, we may also define the dynamic semantics of the core $\mu\tilde{\mu}$-calculus by axioms describing trivial statements about variable binding.

The η_μ and $\eta_{\tilde{\mu}}$ rules state that giving a name to something, and then using it immediately (without repetition) in the same place is the same thing as doing nothing. Additionally, we may say that binding a variable to itself is the same thing as doing nothing:

$$(\mu_\alpha) \qquad \langle \mu\alpha.c \| \alpha\rangle \succ_{\mu_\alpha} c \qquad\qquad (\mu_x) \qquad \langle x \| \tilde{\mu}x.c\rangle \succ_{\tilde{\mu}_x} c$$

These axioms can also be seen as the special cases of $\mu_{\mathcal{S}}$ and $\tilde{\mu}_{\mathcal{S}}$ which are always sound for every strategy, since we always assume that (co-)variables are (co-)values.

If we take the above substitution principles as primitive inference rules like reflexivity, etc. in our equational theory, we can derive $\mu_{\mathcal{S}}$ and $\tilde{\mu}_{\mathcal{S}}$ from the μ_α and $\tilde{\mu}_x$ axioms. The trick is to realize that a command like $\langle V \| \tilde{\mu}x.c\rangle$ is the image of $\langle x \| \tilde{\mu}x.c\rangle$ under substitution of V for x. That is to say that $\langle V \| \tilde{\mu}x.c\rangle$ is syntactically the same as $\langle x \| \tilde{\mu}x.c\rangle \{V/x\}$. Therefore, we can derive the $\tilde{\mu}_{\mathcal{S}}$ axiom from $\tilde{\mu}_x$ and $\mathit{subst}_{\mathcal{S}}$ as follows:

$$\dfrac{\dfrac{}{\langle x \| \tilde{\mu}x.c\rangle = c}\,\tilde{\mu}_x \qquad V \in \mathit{Values}_{\mathcal{S}}}{\langle V \| \tilde{\mu}x.c\rangle = c\{V/x\}}\,\mathit{subst}_{\mathcal{S}}$$

The derivation of $\mu_{\mathcal{S}}$ from μ_α and $\mathit{subst}_{\mathcal{S}}$ is similar:

$$\dfrac{\dfrac{}{\langle \mu\alpha.c \| \alpha\rangle = c}\,\mu_\alpha \qquad E \in \mathit{CoValues}_{\mathcal{S}}}{\langle \mu\alpha.c \| E\rangle = c\{E/\alpha\}}\,\mathit{subst}_{\mathcal{S}}$$

Conversely, the substitution principles are derivable from the more powerful $\tilde{\mu}_{\mathcal{S}}$ and $\mu_{\mathcal{S}}$ axioms. For example, we can derive the $\mathit{subst}_{\mathcal{S}}$ principle for co-values from $\tilde{\mu}_{\mathcal{S}}$ by recognizing that both sides of the equation can be deduced from a command like $\langle V \| \tilde{\mu}x.c\rangle$ with the $\tilde{\mu}_V$ axiom, so that congruence allows us to lift the equality $c = c'$ under the bindings. The full derivation of the co-value $\mathit{subst}_{\mathcal{S}}$ principle is:

$$\dfrac{\dfrac{\dfrac{V \in \mathit{Values}_{\mathcal{S}}}{\langle V \| \tilde{\mu}x.c\rangle = c\{V/x\}}\,\tilde{\mu}_{\mathcal{S}}}{c\{V/x\} = \langle V \| \tilde{\mu}x.c\rangle}\,\mathit{symm} \qquad \dfrac{\dfrac{c = c'}{\langle V \| \tilde{\mu}x.c\rangle = \langle V \| \tilde{\mu}x.c'\rangle}\,\mathit{comp} \qquad \dfrac{V \in \mathit{Values}_{\mathcal{S}}}{\langle V \| \tilde{\mu}x.c'\rangle = c'\{V/x\}}\,\tilde{\mu}_{\mathcal{S}}}{\langle V \| \tilde{\mu}x.c\rangle = c'\{V/x\}}\,\mathit{trans}}{c\{V/x\} = c'\{V/x\}}\,\mathit{trans}$$

and the $\mathit{subst}_{\mathcal{S}}$ principle for co-values may be derived similarly. Therefore, the $\tilde{\mu}_{\mathcal{S}}$ and $\mu_{\mathcal{S}}$ rules may also be seen as a realization of two dual substitution principles of an equational theory in the form of axioms. And furthermore, by controlling substitution we control evaluation itself. *End remark* 5.2.

The Essence of Connectives: Data and Co-Data

When considering a variety of different polarized connectives (Zeilberger, 2008b, 2009; Curien & Munch-Maccagnoni, 2010; Munch-Maccagnoni, 2013), we find that they all fit into one of two dual patterns. Each polarized connective is either positive or negative: positive connectives (following the verificationist approach) describe how to construct terms, whereas negative connectives (following the pragmatist approach) describe how to construct co-terms. In response, both approaches define their other half by inversion, or cases on the allowed patterns of construction. Thus, we use verificationist approach to represent (algebraic) *data types* from functional languages, whose objects are produced by specific constructions and consumed by inversion on the possible constructions. Contrastingly, we use pragmatist approach to represent the dual form of *co-data types*, whose observations, or messages, are described by specific constructions and whose objects respond by inversion on those possible observations.

To study types in the sequent calculus, we will mirror the way that modern programming languages let the user define new types. Functional languages allow for user-defined data types, which are declared by describing the constructors used to build objects of that type. Object-oriented languages allow for user-defined co-data types as interfaces, which are declared by describing the methods (observations) to which objects of that type respond. As we have seen, the sequent calculus unifies these two computational uses of types, letting us describe both user-defined data and co-data types as mirror images of one another. Thus, we aim to encompass all the previously considered connectives as user-defined (co-)data types.

As a starting point, we base the syntax for declaring new user-defined (co-)data type declarations in the sequent calculus on data type declarations in functional languages. However, because the form of (co-)data types in the classical sequent calculus is more expressive than data types in functional languages, we need a syntax that is more general than the usual form of data type declaration from ML-based languages. Therefore, we will look at how the generalized syntax for GADTs in Haskell (Peyton Jones *et al.*, 2006; Schrijvers *et al.*, 2009) may be used for ordinary data type declarations. For example, the typical sum type Either and pair type Both may be declared as:

$$\begin{aligned} &\textbf{data}\,\mathsf{Either}\,a\ b\,\textbf{where} \\ &\qquad \mathsf{Left} : a \to \mathsf{Either}\,a\ b \\ &\qquad \mathsf{Right} : b \to \mathsf{Either}\,a\ b \end{aligned} \qquad \begin{aligned} &\textbf{data}\,\mathsf{Both}\,a\ b\,\textbf{where} \\ &\qquad \mathsf{Pair} : a \to b \to \mathsf{Both}\,a\ b \end{aligned}$$

In the declaration for Either, we specify that there are two constructors: a Left constructor that takes a value of type a and builds a value of type $\mathsf{Either}\,a\ b$, and similarly a Right constructor that take a value of type b and builds a value of type $\mathsf{Either}\,a\ b$. In the declaration for Both, we specify that there is one constructor, Pair, that takes a value of type a, a value of type b, and builds a value of type $\mathsf{Both}\,a\ b$.

When declaring a new type in the sequent calculus, we will take the basic GADT form, but instead describe the constructors with a sequent judgment rather than a function type. For connectives following the verificationist approach, we have data type declarations that introduce new concrete terms and abstract co-terms. For instance, we can give a declaration of $A \oplus B$ as:

$$\begin{aligned} &\textbf{data}\,X \oplus Y\,\textbf{where} \\ &\qquad \iota_1 : X \vdash X \oplus Y \mid \\ &\qquad \iota_2 : Y \vdash X \oplus Y \mid \end{aligned}$$

where we replace the function arrow ($\to$) with logical entailment ($\vdash$), to emphasize that the function type is not inherently baked into the system. Additionally, we mark the distinguished output of each constructor as $X \oplus Y \mid$, which denotes the type of the result produced as the output of the constructed term. This declaration extends the syntax of the language with two new concrete terms for the constructors, $\iota_1\,(v)$ and $\iota_2\,(v)$, and with one new abstract co-term for case analysis, $\tilde{\mu}[\iota_1\,(x).c_1 \mid \iota_2\,(y).c_2]$. Note that these are exactly the system L terms and co-terms for the type $A \oplus B$ from Figure 4.3.

Similarly, we can declare pair types $A \otimes B$ as:

$$\begin{aligned} &\textbf{data}\,X \otimes Y\,\textbf{where} \\ &\qquad (\_,\_) : X, Y \vdash X \otimes Y \mid \end{aligned}$$

where the multiple inputs to the constructor are given as a list of inputs on the left of the sequent, as opposed to the "curried" style used in the declaration of Both. Note that we make use of mix-fix notation $(\_,\_)$ used in functional languages like Agda for describing the constructor syntax, so that this declaration extends the syntax of the language with one new concrete term for the constructor, (v, v'), and one new abstract co-term for case analysis, $\tilde{\mu}[(x, y).c]$. Again, these are exactly the same terms and co-terms for the type $A \otimes B$ in system L.

However, note that user-defined types in the sequent calculus are more general than in functional programming languages. For example, we can declare the positive form of negation as:

$$\begin{gathered}\textbf{data}\, {\sim} X \,\textbf{where} \\ {\sim} : \ \vdash {\sim} X \mid X\end{gathered}$$

where we have an additional output beside the normal distinguished output of type ${\sim}X$, which is not expressible in functional programming languages. This declaration extends the syntax of the language with one new concrete term for the constructor, ${\sim}(e)$, and one new abstract co-term for case analysis, $\tilde{\mu}[{\sim}(\alpha).c]$.

Besides data declarations, we also have co-data declarations that introduce abstract terms and concrete co-terms. We can think of a co-data declaration as an interface that describes the messages understood by an abstract value. By analogy to object-oriented programming, an interface (co-data type declaration) describes the fixed set of methods (co-structures) that an object (case abstraction) has to support (provide cases for), and the object value (case abstraction) defines the behavior that results from a method call (command). For example, we can declare product types $A \,\&\, B$ as:

$$\begin{gathered}\textbf{codata}\, X \,\&\, Y \,\textbf{where} \\ \pi_1 : \ \mid X \,\&\, Y \vdash X \\ \pi_2 : \ \mid X \,\&\, Y \vdash Y\end{gathered}$$

where instead of a distinguished output, we have a distinguished input marked as $\mid A \,\&\, B$ for each co-constructor, which denotes the type of the input expected by the constructed co-term. This declaration extends the language with a new abstract term for case analysis, $\mu(\pi_1\,[\alpha].c_1 | \pi_2\,[\beta].c_2)$, and two concrete co-terms, $\pi_1\,[e]$ and $\pi_2\,[e]$.

Note that these are exactly the terms and co-terms for the type $A \mathbin{\&} B$ as described in system L.

Of note, we find that function types, which are usually baked into functional programming languages as non-definable types, are just another instance of user-defined co-data types in the sequent calculus. In particular, we can declare function types $A \to B$ as:

$$\mathbf{codata}\, X \to Y\, \mathbf{where}$$

$$\_ \cdot \_ : X \mid X \to Y \vdash Y$$

Following the pattern by rote, this declaration extends the language with a new abstract term, $\mu([x \cdot \alpha].c)$ where we put brackets around the call-stack pattern $x \cdot \alpha$ for clarity, and a new concrete co-term, $v \cdot e$. Even though presentation for objects of the function type differs from the usual λ-based presentation, both presentations are mutually definable as syntactic sugar based on one another, as we saw in Chapter IV:

$$\lambda x.v \triangleq \mu(x \cdot \alpha.\langle v \| \alpha \rangle) \qquad\qquad \mu([x \cdot \alpha].c) \triangleq \lambda x.\mu\alpha.c$$

The rest of the basic connectives, including negation and the corresponding unit types for $\oplus$, $\otimes$, $\&$, and $⅋$, are declared as user-defined (co-)data types in Figure 5.6.

Now that we have shown how each of the basic connectives can be described by a data or co-data declaration, our goal is to generalize the pattern to arbitrary, user-defined data and co-data types. First, we introduce the general untyped syntax for arbitrary data and co-data in Figure 5.7.[2] In addition to the expressions inherited from the core $\mu\tilde{\mu}$-calculus, we now have two new forms of terms and co-terms. On the one hand, we have *data structure* terms $\mathsf{K}(\overrightarrow{e}, \overrightarrow{v})$ that build a concrete construction with the *constructor* K, and these may be analysed by a *data case abstraction* co-term $\tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\overrightarrow{\alpha}, \overrightarrow{x}).c}\Big]$ which defines several alternative responses to its given answer matching specific patterns. On the other hand, we have *co-data structure* co-terms $\mathsf{O}[\overrightarrow{v}, \overrightarrow{e}]$ that build a concrete observation with the *observer* O, and these may be analysed by a *co-data case abstraction* term $\mu\Big(\overrightarrow{\mathsf{O}[\overrightarrow{x}, \overrightarrow{\alpha}].c}\Big)$ which defines several alternative responses to its given question matching specific patterns. Note that for both data and co-data case abstractions, we impose the additional syntactic side-condition that the listed

[2] We maintain the same convention from Chapter III and IV for user-defined data and co-data types, whereby terms and co-terms are syntactically distinguished by the use of round parenthesis for terms and square brackets for co-terms.

$$\begin{array}{ll}
\textbf{data}\, X \oplus Y \,\textbf{where} & \textbf{codata}\, X \mathbin{\&} Y \,\textbf{where} \\
\quad \iota_1 : X \vdash X \oplus Y \mid & \quad \pi_1 : \ \mid X \mathbin{\&} Y \vdash X \\
\quad \iota_2 : Y \vdash X \oplus Y \mid & \quad \pi_2 : \ \mid X \mathbin{\&} Y \vdash Y \\
 & \\
\textbf{data}\, X \otimes Y \,\textbf{where} & \textbf{codata}\, X \parr Y \,\textbf{where} \\
\quad (\_,\_) : X, Y \vdash X \otimes Y \mid & \quad [\_,\_] : \ \mid X \parr Y \vdash X, Y \\
 & \\
\textbf{data}\, 0 \,\textbf{where} & \textbf{codata}\, \top \,\textbf{where} \\
 & \\
\textbf{data}\, 1 \,\textbf{where} & \textbf{codata}\, \bot \,\textbf{where} \\
\quad () : \ \vdash 1 \mid & \quad [] : \ \mid \bot \vdash \\
 & \\
\textbf{data}\, X - Y \,\textbf{where} & \textbf{codata}\, A \to B \,\textbf{where} \\
\quad \_ \cdot \_ : X \vdash X - Y \mid Y & \quad \_ \cdot \_ : X \mid X \to Y \vdash Y \\
 & \\
\textbf{data}\, {\sim} X \,\textbf{where} & \textbf{codata}\, \neg X \,\textbf{where} \\
\quad {\sim} : \ \vdash {\sim} X \mid X & \quad \neg : X \mid \neg X \vdash
\end{array}$$

FIGURE 5.6. Declarations of the basic data and co-data types.

$$\begin{array}{rlrl}
x, y, z \in & \mathit{Variable} ::= \dots & \alpha, \beta, \gamma \in & \mathit{CoVariable} ::= \dots \\
\mathsf{K} \in & \mathit{Constructor} ::= \dots & \mathsf{O} \in & \mathit{Observer} ::= \dots
\end{array}$$

$$\begin{aligned}
c \in \mathit{Command} &::= \langle v \| e \rangle \\
v \in \mathit{Term} &::= x \mid \mu\alpha.c \mid \mathsf{K}(\vec{e}, \vec{v}) \mid \mu\Big(\overrightarrow{\mathsf{O}[\vec{x}, \vec{\alpha}].c}\Big) \\
e \in \mathit{CoTerm} &::= \alpha \mid \tilde{\mu}x.c \mid \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c}\Big] \mid \mathsf{O}[\vec{v}, \vec{e}]
\end{aligned}$$

FIGURE 5.7. Adding data and co-data to the core $\mu\tilde{\mu}$ sequent calculus.

constructors $\mathsf{K}, \ldots$ of a data case abstraction are all distinct from one another and likewise the listed observers $\mathsf{O}, \ldots$ of a co-data case abstraction are all distinct.

Second, we give the type system accommodating the general form of declarations for a generic data type constructor F and co-data type constructor G in Figure 5.8. The type constructors in such declarations may connect a sequence of other types, which are represented by the sequence of type variables $\overrightarrow{X}$. Furthermore, a data type may have several constructors, named K_1 to K_n, and a co-data type may have several *observers* which are co-constructors, named O_1 to O_n. The form of these (co-)constructors (i.e. their arity and the type of terms and co-terms they are built from) are described by an arbitrary sequent in the declaration, with the (co-)data being defined in the distinguished input or output position of the sequent. For each such data and co-data declaration, we have additional typing rules for the newly declared connectives which are also shown in Figure 5.8. Because the meaning of a particular type constructor F or G depends on its declaration, we annotate the sequent with the global environment $\mathcal{G}$ that specifies the declarations for all the type constructors, so that $\mathcal{G}$ is used to determine the shape of their left and right logical rules. While these generalized typing rules are involved, they are described in such a way that they exactly replicate the expected typing rules for existing (co-)data types. For instance, by instantiating the generalized typing rules to the basic (co-)data types from Figure 5.6, we recover exactly the same (unpolarized) logical rules from system L in Figure 4.3. Thus, the syntax and typing rules for user-defined (co-)data types subsume each basic connective.

Since we have extended the core $\mu\tilde{\mu}$-calculus syntax with (co-)data structures and abstractions, we must also update the core strategies from Section 5.1 to account for the new values and co-values introduced by the declarations. We could define the (co-)values of each newly declared (co-)data type on a case-by-case basis. However, instead we can also to define the (co-)values of (co-)data types generically across all declarations, which besides being more economical prevents ad-hoc decisions. To do this, we define a strategy $\mathcal{S}$ once and for all over an untyped syntax that was given in Figure 5.7 which already accounts for all possible (co-)data type declarations. Also note that the notion of evaluation context does not change with the addition of (co-)data, so we only need to consider how the substitution strategy is impacted. Thus, a strategy can be given for all possible extensions of newly-declared (co-)data types by carving out a set of values and co-values from the untyped syntax of terms and co-terms.

$$A, B, C \in \mathit{Type} ::= X \mid \mathsf{F}(\overrightarrow{A}) \quad X, Y, Z \in \mathit{TypeVariable} ::= \dots \quad \mathsf{F}, \mathsf{G} \in \mathit{Connective} ::= \dots$$

$$\mathit{decl} \in \mathit{Declaration} ::= \mathbf{data}\,\mathsf{F}(\overrightarrow{X})\,\mathbf{where}\,\overrightarrow{\mathsf{K} : \overrightarrow{A} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B}}$$

$$\mid \mathbf{codata}\,\mathsf{G}(\overrightarrow{X})\,\mathbf{where}\,\overrightarrow{\mathsf{O} : \overrightarrow{A} \mid \mathsf{G}(\overrightarrow{X}) \vdash \overrightarrow{B}}$$

$$\mathcal{G} \in \mathit{GlobalEnv} ::= \overrightarrow{\mathit{decl}} \qquad \Gamma \in \mathit{InputEnv} ::= \overrightarrow{x : A} \qquad \Delta \in \mathit{OutputEnv} ::= \overrightarrow{\alpha : A}$$

$$J, H \in \mathit{Judgement} ::= c : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right) \mid (\Gamma \vdash_{\mathcal{G}} v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash_{\mathcal{G}} \Delta)$$

Core rules:

$$\frac{}{x : A \vdash_{\mathcal{G}} x : A \mid}\ \mathit{VR} \qquad \frac{}{\mid \alpha : A \vdash_{\mathcal{G}} \alpha : A}\ \mathit{VL}$$

$$\frac{c : \left(\Gamma \vdash_{\mathcal{G}} \alpha : A, \Delta\right)}{\Gamma \vdash_{\mathcal{G}} \mu\alpha.c : A \mid \Delta}\ \mathit{AR} \qquad \frac{c : \left(\Gamma, x : A \vdash_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}x.c : A \vdash_{\mathcal{G}} \Delta}\ \mathit{AL}$$

$$\frac{\Gamma \vdash_{\mathcal{G}} v : A \mid \Delta \quad \Gamma' \mid e : A \vdash_{\mathcal{G}} \Delta'}{\langle v \| e \rangle : \left(\Gamma', \Gamma \vdash_{\mathcal{G}} \Delta', \Delta\right)}\ \mathit{Cut}$$

Logical rules:

Given $\mathbf{data}\,\mathsf{F}(\overrightarrow{X})\,\mathbf{where}\,\overrightarrow{\mathsf{K}_i : \overrightarrow{A_{ij}}^j \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij}}^j}^i \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{\Gamma'_j \mid e : B_{ij}\overrightarrow{\{C/X\}} \vdash_{\mathcal{G}} \Delta'_j}^j \quad \overrightarrow{\Gamma_j \mid v : A_{ij}\overrightarrow{\{C/X\}} \vdash_{\mathcal{G}} \Delta_j}^j}{\overrightarrow{\Gamma_j}^j, \overrightarrow{\Gamma'_j}^j \vdash_{\mathcal{G}} \mathsf{K}_i(\overrightarrow{e}, \overrightarrow{v}) : \mathsf{F}(\overrightarrow{C}) \mid \overrightarrow{\Delta_j}^j, \overrightarrow{\Delta'_j}^j}\ \mathsf{F}R_{\mathsf{K}_i}$$

$$\frac{\overrightarrow{c_i : \left(\Gamma, \overrightarrow{x_i : A_i\overrightarrow{\{C/X\}}} \vdash_{\mathcal{G}} \overrightarrow{\alpha_i : B_i\overrightarrow{\{C/X\}}}, \Delta\right)}^i}{\Gamma \mid \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_i}, \overrightarrow{x_i}).c_i}^i\right] : \mathsf{F}(\overrightarrow{C}) \vdash_{\mathcal{G}} \Delta}\ \mathsf{F}L$$

Given $\mathbf{codata}\,\mathsf{G}(\overrightarrow{X})\,\mathbf{where}\,\overrightarrow{\mathsf{O}_i : \overrightarrow{A_{ij}}^j \mid \mathsf{G}(\overrightarrow{X}) \vdash \overrightarrow{B_{ij}}^j}^i \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{c_i : \left(\Gamma, \overrightarrow{x_i : A_i\overrightarrow{\{C/X\}}} \vdash_{\mathcal{G}} \overrightarrow{\alpha_i : B_i\overrightarrow{\{C/X\}}}, \Delta\right)}^i}{\Gamma \vdash_{\mathcal{G}} \mu\left(\overrightarrow{\mathsf{O}_i[\overrightarrow{x_i}, \overrightarrow{\alpha_i}].c_i}^i\right) : \mathsf{G}(\overrightarrow{C}) \mid \Delta}\ \mathsf{G}R$$

$$\frac{\overrightarrow{\Gamma_j \mid v : A_{ij}\overrightarrow{\{C/X\}} \vdash_{\mathcal{G}} \Delta_j}^j \quad \overrightarrow{\Gamma'_j \mid e : B_{ij}\overrightarrow{\{C/X\}} \vdash_{\mathcal{G}} \Delta'_j}^j}{\overrightarrow{\Gamma_j}^j, \overrightarrow{\Gamma'_j}^j \mid \mathsf{O}_i[\overrightarrow{v}, \overrightarrow{e}] : \mathsf{G}(\overrightarrow{C}) \vdash_{\mathcal{G}} \overrightarrow{\Delta_j}^j, \overrightarrow{\Delta'_j}^j}\ \mathsf{G}L_{\mathsf{O}_i}$$

FIGURE 5.8. Types of declared (co-)data in the parametric $\mu\tilde{\mu}$ sequent calculus.

Our call-by-value strategy $\mathcal{V}$ will mimic ML-like languages. Therefore, we can say that a data structure is a value of $\mathcal{V}$ when all of its sub-terms are values. For example, a pair (v_1, v_2) is a value when both v_1 and v_2 are values, and an injection, $\iota_1\,(v)$ or $\iota_2\,(v)$, is a value when v is a value. Additionally, all co-data case abstractions (i.e. objects) are considered values. This comes from the fact that a λ-abstraction, which we represent as a case abstraction, is a value in the call-by-value λ-calculus. As before, though, we continue to admit every single co-term as a co-value. Thus, we achieve the $\mathcal{V}$ strategy with arbitrary (co-)data types shown in Figure 5.9.

Our call-by-name strategy $\mathcal{N}$ will mimic call-by-name λ-calculi with data types, similar to Haskell. Therefore, we still admit every single term as a value. The co-values of $\mathcal{N}$ represent "strict" contexts from a call-by-name λ-calculus. For example, case analysis is always strict in these languages, therefore the case abstraction of a data type is a co-value. Additionally, an observation of a co-data type is a co-value when all sub-(co-)terms are (co-)values. This follows the definition of co-values from the call-by-name half of the dual calculi from Section 3.3 as well as the hereditary nature of strict contexts for functions and products in a call-by-value λ-calculus. For example, the contexts:

$$\mathbf{let}\,x = \square\ 1\,\mathbf{in}\,5 \qquad\qquad \mathbf{let}\,x = \pi_1\ \square\,\mathbf{in}\,4$$

is not strict because x is not required to compute the result 5, even though we are applying the hole $\square$ to an argument or projecting out one of its components. However, the contexts:

$$\mathbf{case}\,\square\ 1\,\mathbf{of}\,\iota_1(x) \Rightarrow 5 \mid \iota_2(y) \Rightarrow 10$$
$$\mathbf{case}\,\pi_1\ \square\,\mathbf{of}\,\iota_1(x) \Rightarrow 5 \mid \iota_2(y) \Rightarrow 10$$

are both strict because we need to compute the input plugged into $\square$ to determine which branch to take. Thus, we achieve the $\mathcal{N}$ strategy with arbitrary (co-)data types shown in Figure 5.9.

Finally, our call-by-need strategy $\mathcal{LV}$ is the most complex, since it accounts for the memoization used to efficiently implement lazy evaluation for the Haskell language. Intuitively, the key to understanding call-by-need is to think about sharing, where the values and co-values of $\mathcal{LV}$ represent terms and co-terms that may be freely copied as many times as necessary. In $\mathcal{LV}$, a structure can be copied if all of its sub-

$$\begin{aligned}
V \in \mathit{Value}_{\mathcal{V}} &::= x \mid \mathsf{K}(\vec{e}, \vec{V}) \mid \mu\Big(\overrightarrow{\mathsf{O}[\vec{x}, \vec{\alpha}].c}\Big) \\
E \in \mathit{CoValue}_{\mathcal{V}} &::= e \\
V \in \mathit{Value}_{\mathcal{N}} &::= v \\
E \in \mathit{CoValue}_{\mathcal{N}} &::= \alpha \mid \mathsf{O}[\vec{v}, \vec{E}] \mid \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c}\Big]
\end{aligned}$$

FIGURE 5.9. Call-by-value ($\mathcal{V}$) and call-by-name ($\mathcal{N}$) substitution strategies extended with arbitrary (co-)data types.

$$\begin{aligned}
V \in \mathit{Value}_{\mathcal{LV}} &::= x \mid \mathsf{K}(\vec{E}, \vec{V}) \mid \mu\Big(\overrightarrow{\mathsf{O}[\vec{x}, \vec{\alpha}].c}\Big) \\
E \in \mathit{CoValue}_{\mathcal{LV}} &::= \alpha \mid \tilde{\mu}x.D[\langle x \| E \rangle] \mid \mathsf{O}[\vec{v}, \vec{E}] \mid \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c}\Big] \\
V \in \mathit{Value}_{\mathcal{LN}} &::= x \mid \mu\alpha.D[\langle V \| \alpha \rangle] \mid \mathsf{K}(\vec{e}, \vec{V}) \mid \mu\Big(\overrightarrow{\mathsf{O}[\vec{x}, \vec{\alpha}].c}\Big) \\
E \in \mathit{CoValue}_{\mathcal{LN}} &::= \alpha \mid \mathsf{O}[\vec{V}, \vec{E}] \mid \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c}\Big]
\end{aligned}$$

FIGURE 5.10. “Lazy-call-by-value” ($\mathcal{LV}$) and “lazy-call-by-name” ($\mathcal{LN}$) substitution strategies extended with arbitrary (co-)data types.

(co-)terms can be copied, following the usual treatment of sharing for data structures in implementations of Haskell. Additionally, a case abstraction can always be copied, following the treatment of λ-abstractions in implementations of Haskell. Thus, we achieve the $\mathcal{LV}$ strategy with arbitrary (co-)data types shown in Figure 5.10. The dual lazy call-by-name strategy $\mathcal{LN}$ is also shown in Figure 5.10, which is derived by exchanging the role of terms and co-terms from $\mathcal{LV}$.

Evaluating Data and Co-Data

Having resolved the fundamental dilemma of computation in the parametric $\mu\tilde{\mu}$-calculus via a variety of strategies, and having extended the language with new syntactic forms for user-defined (co-)data types, we now need to explain how the constructs of (co-)data types behave. To that end, we introduce two different semantics for (co-)data in the parametric sequent calculus:

- a *typed* $\beta\eta$ theory that is independent of the chosen strategy, and
- an *untyped* $\beta\varsigma$ theory that depends on the chosen strategy.

Both of these theories have their own advantages and disadvantages. On the one hand, the $\beta\eta$ theory gives a canonical definition of the dynamic semantics of (co-)data independently of any evaluation strategy, but relies on types to do so sensibly. On the other hand, the $\beta\varsigma$ theory gives a mechanism for running programs without resorting to types and equational reasoning, but it depends on the chosen evaluation strategy and relates fewer programs than $\beta\eta$.

The typed $\beta\eta$ theory of (co-)data

Since the evaluation strategy is handled by the equational theory of the core $\mu\tilde{\mu}$-calculus, we should express the behavior of (co-)data type structures in some way that is valid for *any* choice of strategy, $\mathcal{S}$. In other words, given a set of data and co-data type declarations $\mathcal{G}$, we would like to describe the equational theory for the language extended with those types. As we saw in Chapter II, in the λ-calculus the dynamic meaning of types are expressed by β and η laws. The β laws characterize the main computational force of a type, whereas the η laws characterize a form of extensionality for a type. Therefore, to accomplish our goal in the sequent calculus, we will use an analogous form of β and η laws for defining the dynamic meaning of user-defined (co-)data types, and like in the λ-calculus, the η laws must be typed to be sensible.

For example, we may extend the equational theory with the following β law for functions:

$$(\beta^{\rightarrow}) \qquad \langle \mu([x \cdot \alpha].c) \| v \cdot e \rangle \succ_{\beta^{\rightarrow}} \langle v \| \tilde{\mu}x.\, \langle \mu\alpha.c \| e \rangle \rangle$$

which matches on the structure of a function call and binds the sub-components to the appropriate (co-)variables. Notice that this rule applies for *any* function call, $v \cdot e$, whether or not v or e are (co-)values, so $\beta^{\rightarrow}$ does not depend on any substitution strategy. This works because we avoid performing substitution in the $\beta^{\rightarrow}$ axiom, and instead v and e are put in interaction with input and output abstractions. Since we have already informed the core structural theory about our chosen strategy, we know that the substitutions will be performed in the correct order. Therefore, if we are evaluating our program according to call-by-value, we would have to evaluate v first (via the $\mu_{\mathcal{S}}$ rule if necessary) before substituting for x. Likewise, in call-by-name, we would have to evaluate e first (via the $\tilde{\mu}\mathcal{S}$ rule if necessary) before substituting for α.

Next, we have the following η law for functions:

$$(\eta^{\to}) \qquad\qquad z : A \to B \prec_{\eta^{\to}} \mu([x \cdot \alpha].\langle z \| x \cdot \alpha \rangle)$$

which says that an unknown function, z, is equivalent to a trivial case abstraction that matches a function call and forwards it along, unchanged, to z. Here, we use the variable z to stand in for an unknown value, since we are only allowed to substitute values for variables.

Note that the more general but strategy-dependent presentation of the η law, which applies to an arbitrary value rather than just a variable, is derivable from the more restrictive $\eta^{\to}$ law above and the equational theory of substitution in the parametric $\mu\tilde{\mu}$-calculus:

$$\begin{aligned}
V : A \to B &=_{\eta_\mu} \mu\gamma.\, \langle V \| \gamma \rangle \\
&=_{\eta_{\tilde{\mu}}} \mu\gamma.\, \langle V \| \tilde{\mu}z.\, \langle z \| \gamma \rangle \rangle \\
&=_{\eta^{\to}} \mu\gamma.\, \langle V \| \tilde{\mu}z.\, \langle \mu([x \cdot \alpha].\langle z \| x \cdot \alpha \rangle) \| \gamma \rangle \rangle \\
&=_{\tilde{\mu}_{\mathcal{S}}} \mu\gamma.\, \langle \mu([x \cdot \alpha].\langle V \| x \cdot \alpha \rangle) \| \gamma \rangle \\
&=_{\eta_\mu} \mu([x \cdot \alpha].\langle V \| x \cdot \alpha \rangle)
\end{aligned}$$

This has the nice side effect that neither the $\beta^{\to}$ or $\eta^{\to}$ rules themselves explicitly mention values or co-values in any way—they are strategy independent.[3]

Remark 5.3. To make the comparison with previous characterizations of functions in the sequent calculus from Chapter III, we can be more formal about the relationship between λ-abstractions and co-case abstractions over call stacks. In particular, taking the round trip of the mutual syntactic sugar definitions presented in Section 5.2 results in equal (co-)terms:

$$\lambda x.v \triangleq \mu([x \cdot \alpha].\langle v \| \alpha \rangle) \triangleq \lambda x.\mu\alpha.\, \langle v \| \alpha \rangle =_{\eta_\mu} \lambda x.v$$

$$\mu([x \cdot \alpha].c) \triangleq \lambda x.\mu\alpha.c \triangleq \mu([x \cdot \alpha].\langle \mu\alpha.c \| \alpha \rangle) =_{\mu_{\mathcal{S}}} \mu([x \cdot \alpha].c)$$

[3] It also has the pleasant effect that the side conditions on the free variables of V used to prevent static variable capture automatically come from capture-avoiding substitution in the equational theory.

where the application of $\mu_{\mathcal{S}}$ is valid for any $\mathcal{S}$, since co-variables are always co-values. We may also rephrase these β and η axioms for functions into the λ-based syntax:

$$(\beta^{\lambda}) \qquad \langle \lambda x.v \| v' \cdot e \rangle \succ_{\beta^{\lambda}} \langle v' \| \tilde{\mu} x. \langle v \| e \rangle \rangle \qquad (\eta^{\lambda}) \qquad z \prec_{\eta^{\lambda}} \lambda x.\mu\alpha. \langle z \| x \cdot \alpha \rangle$$

Note that these are mutually derivable from the $\beta^{\rightarrow}$ and $\eta^{\rightarrow}$ axioms according to the syntactic sugar definition for λ-abstractions, along with the $\mu_{\mathcal{S}}$ and η_{μ} axioms. Thus, the two presentations of functions really are equivalent to one another: we can view a function as a λ-abstraction mapping an input to an output, or as an object that deconstructs an observation in the shape of a call-stack. *End remark* 5.3.

Remark 5.4. Even though we can derive a generalized version of the $\eta^{\rightarrow}$ axiom which applies to values, it is important to note that the $\eta[\rightarrow]$ rule would not work if we replaced z with a general term v. The exact same problem occurs in the call-by-value λ-calculus, where we admit non-terminating terms. If we are allowed to η expand any term, then we have the equality:

$$5 =_{\beta} (\lambda x.5)\ (\lambda y.\Omega\ y) =_{\eta} (\lambda x.5)\ \Omega \approx \Omega$$

where Ω stands in for a term that loops forever. So if we allow η expansion of arbitrary terms in the call-by-value λ-calculus, then a value like 5 is the same thing as a program that loops forever. The solution in the call-by-value λ-calculus is to limit the η rule to only apply to values. It should then be no surprise that the same limitation is necessary for the analogous $\eta^{\rightarrow}$ axiom in the classical sequent calculus, where we can always have the term $\mu\_.c$ that never returns a result just like an infinite loop. *End remark* 5.4.

Similarly, we can explain the behavior of the co-data type for products with an analogous set of β and η axioms. The $\beta^{\&}$ axiom demonstrates how an object of $A \& B$ matches on the structure of projection, binding the consumer for its output to the appropriate co-variable:

$$(\beta^{\&}) \qquad \langle \mu(\pi_1\,[\alpha].c_1 \mid \pi_2\,[\beta].c_2) \| \pi_1\,[e] \rangle \succ_{\beta^{\&}} \langle \mu\alpha.c_1 \| e \rangle$$
$$(\beta^{\&}) \qquad \langle \mu(\pi_1\,[\alpha].c_1 \mid \pi_2\,[\beta].c_2) \| \pi_2\,[e] \rangle \succ_{\beta^{\&}} \langle \mu\beta.c_2 \| e \rangle$$

Again, this rule is safe for *any* projection $\pi_1[e]$ or $\pi_2[e]$ because the underlying co-term e is put in interaction with an output abstraction, so that the substitution is performed only in the correct situation. Likewise, the $\eta^{\&}$ axiom states that an unknown product value z is equivalent to a redundant co-case analysis which forwards its output to z:

$$(\eta^{\&}) \qquad z : A \mathbin{\&} B \prec_{\eta^{\&}} \mu(\pi_1\,[\alpha].\langle z \| \pi_2\,[\alpha]\rangle \mid \pi_2\,[\beta].\langle z \| \pi_2\,[\beta]\rangle)$$

In other words, the variable z, which stands in for some object of $A \mathbin{\&} B$, must be equivalent to an object with the same response to the π_1 and π_2 projections. As before, we have the generalized, strategy-dependent version of the $\eta^{\&}$ as an equality:

$$V : A \mathbin{\&} B = \mu(\pi_1\,[\alpha].\langle V \| \pi_1\,[\alpha]\rangle \mid \pi_2\,[\beta].\langle V \| \pi_2\,[\beta]\rangle) \qquad \alpha, \beta \notin FV(V)$$

which is derivable from $\eta^{\&}$, η_μ, $\eta_{\tilde{\mu}}$, and $\tilde{\mu}_{\mathcal{S}}$, meaning that the only thing that is observable about an object of $A \mathbin{\&} B$ is its response to observations of the form $\pi_1\,[\alpha]$ and $\pi_2\,[\beta]$.

The β and η laws for user-defined data types follow a similar, but mirrored, pattern. For example, the β rules for $\oplus$ is exactly dual to $\beta^{\&}$, and performs case analysis on the tag of the term without requiring that the sub-term be a value:

$$\begin{aligned}
&(\beta^{\oplus}) && \langle \iota_1\,(v) \| \tilde{\mu}[\iota_1\,(x).c_1 \mid \iota_2\,(y).c_2]\rangle \succ_{\beta^{\oplus}} \langle v \| \tilde{\mu}x.c_1\rangle \\
&(\beta^{\oplus}) && \langle \iota_2\,(v) \| \tilde{\mu}[\iota_1\,(x).c_1 \mid \iota_2\,(y).c_2]\rangle \succ_{\beta^{\oplus}} \langle v \| \tilde{\mu}y.c_2\rangle
\end{aligned}$$

These rules work for *any* injected terms $\iota_1\,(v)$ or $\iota_2\,(v)$ because they put the sub-term v in interaction with an input abstraction, allowing the equational theory of the underlying structural core take care of managing evaluation order. For example, while this rule is stronger than the one given for the call-by-value half of the dual sequent calculi from Section 3.3, it is still valid according to its Wadler's (2003) call-by-value continuation-passing style (CPS) transformation. The η rule for $\oplus$ is also dual to $\eta^{\&}$, where we expand an unknown co-value γ into a case abstraction:

$$(\eta^{\oplus}) \qquad \gamma : A \oplus B \prec_{\eta^{\oplus}} \tilde{\mu}[\iota_1\,(x).\langle \iota_1\,(x) \| \gamma\rangle \mid \iota_2\,(y).\langle \iota_2\,(y) \| \gamma\rangle]$$

Thus, the only thing that matters for an unknown sum co-value γ is the way that it responds to an input of the form $\iota_1\,(x)$ or $\iota_2\,(x)$.

As a final example, consider the β axiom for pairs, which matches on the structure of the pair and binds the sub-terms to the appropriate variables:

$$(\beta^{\otimes}) \qquad \langle (v, v') \| \tilde{\mu}[(x, y).c]\rangle \succ_{\beta^{\otimes}} \langle v \| \tilde{\mu}x.\, \langle v' \| \tilde{\mu}y.c\rangle\rangle$$

$$
\begin{array}{ll}
(\beta^{\mathsf{F}}) & \langle \mathsf{K}_i(\vec{e}, \vec{v}) \| \tilde{\mu}[\cdots \mid \mathsf{K}_i(\vec{\alpha}, \vec{x}).c_i \mid \cdots] \rangle \succ_{\beta^{\mathsf{F}}} \langle \mu\vec{\alpha}.\, \langle \vec{v} \| \tilde{\mu}\vec{x}.c_i \rangle \| \vec{e} \rangle \\
(\beta^{\mathsf{G}}) & \langle \mu(\cdots \mid \mathsf{O}_i[\vec{x}, \vec{\alpha}].c_i \mid \cdots) \| \mathsf{O}_i[\vec{v}, \vec{e}] \rangle \succ_{\beta^{\mathsf{G}}} \langle \vec{v} \| \tilde{\mu}\vec{x}.\, \langle \mu\vec{\alpha}.c_i \| \vec{e} \rangle \rangle \\
(\eta^{\mathsf{F}}) & \gamma : \mathsf{F}(\vec{C}) \prec_{\eta^{\mathsf{F}}} \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i(\vec{\alpha}, \vec{x}).\langle \mathsf{K}_i(\vec{\alpha}, \vec{x}) \| \gamma \rangle}^{i} \right] \\
(\eta^{\mathsf{G}}) & z : \mathsf{G}(\vec{C}) \prec_{\eta \mathsf{G}} \mu\left(\overrightarrow{\mathsf{O}_i[\vec{x}, \vec{\alpha}].\langle z \| \mathsf{O}_i[\vec{x}, \vec{\alpha}] \rangle}^{i} \right)
\end{array}
$$

FIGURE 5.11. The $\beta\eta$ laws for declared data and co-data types.

The $\beta^{\otimes}$ rule follows the intuition that a destructuring binding on the structure of a known pair is the same thing as binding the sub-terms of the pair one at a time. Next, the η axiom for pairs states that a co-variable γ expecting a pair $A \otimes B$ as input is the same as the redundant case abstraction which breaks apart and re-assembles it input before forwarding it to γ:

$$
(\eta^{\otimes}) \qquad \gamma : A \otimes B \prec_{\eta^{\otimes}} \tilde{\mu}[(x, y).\langle (x, y) \| \gamma \rangle]
$$

We now look to summarize all the β and η laws considered so far into their general form for user-defined (co-)data types. That way, we can take an arbitrary declaration for a user-defined (co-)data type and automatically generate the appropriate axioms to characterize the run-time behavior of its programs. In particular, given the declarations for a generic data type constructor F and co-data type constructor G in Figure 5.8, we show the corresponding β and η axioms in Figure 5.11. Note that these rules use syntactic sugar for writing a sequence of input and output bindings. That is, given a sequence of terms $\vec{v} = v_1, \dots, v_n$ and variables $\vec{x} = x_1, \dots, x_n$, or a sequence of co-terms $\vec{e} = e_1, \dots, e_n$ and co-variables $\vec{\alpha} = \alpha_1, \dots, \alpha_n$, the sequence bindings are defined as:

$$
\langle \vec{v} \| \tilde{\mu}\vec{x}.c \rangle \triangleq \langle v_1 \| \tilde{\mu}x_1. \dots \langle v_n \| \tilde{\mu}x_n.c \rangle \rangle \qquad \langle \mu\vec{\alpha}.c \| \vec{e} \rangle \triangleq \langle \mu\alpha._1 \dots \langle \mu\alpha_n.c \| e_n \rangle \| e_1 \rangle
$$

The type restriction on the η laws are necessary to prevent the associated equational theory from collapsing, similar to the situation in the λ-calculus as discussed in Section 2.2. For example, the nullary case of the η law for co-data gives us $x =_\eta \mu() =_\eta y$ which is fine if both $x : \top$ and $y : \top$, but is troublesome if x and y stand for some other kind of object like functions or products.

$$
\begin{array}{ll}
(\beta_{\mathcal{S}}) & \left\langle \mathsf{K}(\vec{E},\vec{V}) \middle\| \tilde{\mu}[\cdots \mid \mathsf{K}(\vec{\alpha},\vec{x}).c \mid \cdots] \right\rangle \succ_{\beta_{\mathcal{S}}} c\left\{\overrightarrow{E/\alpha}, \overrightarrow{V/x}\right\} \\
(\beta_{\mathcal{S}}) & \left\langle \mu(\cdots \mid \mathsf{O}(\vec{x},\vec{\alpha}).c \mid \cdots) \middle\| \mathsf{O}(\vec{E},\vec{V}) \right\rangle \succ_{\beta_{\mathcal{S}}} c\left\{\overrightarrow{V/x}, \overrightarrow{E/\alpha}\right\}
\end{array}
$$

$$
\left.
\begin{array}{ll}
(\varsigma_{\mathcal{S}}) & \mathsf{K}(\vec{E},e',\vec{e},\vec{v}) \succ_{\varsigma_{\mathcal{S}}} \mu\alpha. \left\langle \mu\beta. \left\langle \mathsf{K}(\vec{E},\beta,\vec{e},\vec{v}) \middle\| \alpha \right\rangle \middle\| e' \right\rangle \\
(\varsigma_{\mathcal{S}}) & \mathsf{K}(\vec{E},\vec{V},v',\vec{v}) \succ_{\varsigma_{\mathcal{S}}} \mu\alpha. \left\langle v' \middle\| \tilde{\mu}y. \left\langle \mathsf{K}(\vec{E},\vec{V},y,\vec{v}) \middle\| \alpha \right\rangle \right\rangle \\
(\varsigma_{\mathcal{S}}) & \mathsf{O}(\vec{V},v',\vec{v},\vec{e}) \succ_{\varsigma_{\mathcal{S}}} \tilde{\mu}x. \left\langle v' \middle\| \tilde{\mu}y. \left\langle x \middle\| \mathsf{O}(\vec{V},y,\vec{v},\vec{e}) \right\rangle \right\rangle \\
(\varsigma_{\mathcal{S}}) & \mathsf{O}(\vec{V},\vec{E},e',\vec{e}) \succ_{\varsigma_{\mathcal{S}}} \tilde{\mu}x. \left\langle \mu\beta. \left\langle x \middle\| \mathsf{O}(\vec{V},\vec{E},\beta,\vec{e}) \right\rangle \middle\| e' \right\rangle
\end{array}
\right\}
\begin{array}{c}
v' \notin \mathit{Values}_{\mathcal{S}} \\
e' \notin \mathit{CoValues}_{\mathcal{S}} \\
x, y, \alpha, \beta \text{ fresh}
\end{array}
$$

FIGURE 5.12. The parametric $\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}$ laws for arbitrary data and co-data.

The untyped $\beta\varsigma$ theory of (co-)data

Next, we consider an alternative semantics for (co-)data in the sequent calculus which is based on system L's strategy-dependent β laws from Figure 4.8 and ς laws from Figure 4.11 in Chapter IV. These rules can be generalized to arbitrary (co-)data structures as shown in Figure 5.12. Both the β and ς perform two separate and non-overlapping duties. The ς laws evaluate unevaluated data and co-data structures by lifting out an unevaluated (i.e. non-(co-)value) sub-expression and giving it a name, so that computation can proceed to determine its (co-)value. The β laws perform pattern-matching on fully-evaluated structures built from (co-)values by substituting the contained (co-)values for the corresponding (co-)variables in the matching pattern of a case abstraction. Note that the strategy-dependent β laws in Figure 5.12 are less general than the strategy-independent ones from Figure 5.11, which can pattern-match on *any* structure, so that they do not accidentally perform the same work of giving names to unevaluated components that would otherwise be done by a ς rule. Also notice that these rewriting rules do not depend on types at all: they function over untyped syntax, letting us evaluate programs without resorting to information about static types.

Besides just being meaningful for executing untyped programs, the $\beta\varsigma$ semantics for (co-)data has another advantage over the $\beta\eta$ semantics: the $\beta\varsigma$ reduction theory is easily *confluent*.

Definition 5.1 (confluence)**.** A reduction relation $\to_R$ in the sequent calculus is *(strongly) confluent* if and only if all divergent reductions $c_1 \twoheadleftarrow_R c \twoheadrightarrow_R c_2$ join together as $c_1 \twoheadrightarrow_R c' \twoheadleftarrow_R c_2$ for some c', and similarly for (co-)terms. Furthermore, a reduction relation $\to_R$ in the sequent calculus is *locally (or weakly) confluent* if and only if all divergent reductions $c_1 \leftarrow_R c \to_R c_2$ join together as $c_1 \twoheadrightarrow_R c' \twoheadleftarrow_R c_2$ for some c', and similarly for (co-)terms.

A well-known consequence of confluence is that, for any (strongly) confluent $\to_R$, the equational theory $=_R$ is the same thing as convertibility $\twoheadrightarrow_R \twoheadleftarrow_R$. That means that in order to determine if two expressions are equal by a confluent theory, we only need to normalize both and compare their normal forms. Unfortunately, even putting issues involving types aside, the combination of the η law with the $\mu\tilde{\mu}$ laws notoriously breaks confluence. For example, if we consider just functions, we have the following critical pair between $\eta^{\to}_{\mathcal{S}}$ (which generalizes $\eta^{\to}$ to values) and $\mu_{\mathcal{S}}$:

$$\mu\_.c \leftarrow_{\eta^{\to}_{\mathcal{S}}} \mu([x \cdot \beta].\langle \mu\_.c \| x \cdot \beta \rangle) \to_{\mu_{\mathcal{S}}} \mu([x \cdot \beta].c)$$

So confluence in the presence of η and $\mu\tilde{\mu}$ is not so straightforward. Contrarily, confluence of the $\beta\varsigma$ theory of (co-)data is straightforwardly confluent when combined with the core $\mu\tilde{\mu}$ theory.

Theorem 5.1 (Parametric confluence)**.** *The $\to_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}}$ reduction relation is confluent for any substitution strategy $\mathcal{S}$ such that $\succ_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}}$ is deterministic and the sets $Value_{\mathcal{S}}$ and $CoValue_{\mathcal{S}}$ are both forward closed under $\to_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}}$.*

Proof. By the decreasing diagrams (van Oostrom, 1994) method of confluence. As shorthand, let $R = \mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}$. Our measure of decreasingness based on increasing *depth* of the context in which reduction occurs, which finds the context of compatibility lifting a basic $\succ_R$ rewrite into $\to_R$. First, we define the depth of a reduction $c_1 \to_R c_2$, denoted by $depth(c_1 \to_R c_2)$, as the height of the hole in the context C from its root such that $c_1 = C[c_1']$, $c_2 = C[c_2']$, and $c_1' \succ_R c_2'$, and similarly for reduction on (co-)terms. This measure is well-founded (i.e. for any set of reductions, there is a minimal one with no others less than it) because the syntax of commands and (co-)terms are finitely deep. Second, we define the measures of strict decreasingness on reductions, written $(c_1 \to_{R_1} c_1') < (c_2 \to_{R_2} c_1')$, as $depth(c_1 \to_{R_1} c_1') > depth(c_2 \to_{R_2} c_2')$, and similarly for (co-)term reductions. The goal of the proof is then to show that for every rule R_1

and R_2 of R giving a divergent pair of reductions $c_1 \leftarrow_{R_1} c \rightarrow_{R_2} c_2$ (and similarly for (co-)terms), the two ends join back together as

$$c_1 \twoheadrightarrow_{R'_1} \underline{\rightarrow}_{R''_2} \twoheadrightarrow_{R'} c' \twoheadleftarrow_{R'} \underline{\leftarrow}_{R''_1} \twoheadleftarrow_{R'_2} c_2$$

where $\underline{\rightarrow}_{R''_i}$ is zero or one R reductions of the same measure as $c \rightarrow_{R_i} c_i$, each $\twoheadrightarrow_{R'_i}$ reduction is less than $c \rightarrow_{R_i} c_i$, and each $\twoheadrightarrow_{R'}$ reduction is less than *either* $c \rightarrow_{R_1} c_1$ or $c \rightarrow_{R_2} c_2$.

We now demonstrate the (strong) confluence of $\rightarrow_R$ by showing that the local confluence diagrams of each diverging pair of $\rightarrow_R$ reductions are all decreasing by the above measure. In the cases where the two diverging reductions are disjoint (i.e. their depths are unordered, so the reductions occur in separate sub-expressions of the overall expression), then they trivially join in one step via compatibility. Otherwise, the two diverging reductions are nested (i.e. their depths are ordered, so that one reduction occurs inside the other or directly on the same expression). In this case, we proceed by cases on rewriting rule used for the outer-most reduction, so the possible nested diverging reductions join back together by decreasing diagrams as follows:

- $\langle \mu\alpha.c \| E \rangle \succ_{\mu_\mathcal{S}} c\{E/\alpha\}$ has four different possible nested reductions:
  - If $\langle \mu\alpha.c \| E \rangle \succ_R c'$ then $c' = c\{E/\alpha\}$, because the only possibility for R is $\tilde{\mu}_\mathcal{S}$, but $\succ_{\mu_\mathcal{S}\tilde{\mu}_\mathcal{S}}$ is deterministic by assumption.
  - If $\langle \mu\alpha.c \| E \rangle \rightarrow_{\eta_\mu} \langle v \| E \rangle$ because $c = \langle v \| \alpha \rangle$ and $\alpha \notin FV(v)$ then $c\{E/\alpha\} = \langle v \| E \rangle$ as well, so the two divergent reductions trivially join.
  - If $\langle \mu\alpha.c \| E \rangle \rightarrow_R \langle \mu\alpha.c' \| E \rangle$ because $c \rightarrow_R c'$ then

    $$c\{E/\alpha\} \rightarrow_R c'\{E/\alpha\} \prec_{\mu_\mathcal{S}} \langle \mu\alpha.c' \| E \rangle$$

    because $\rightarrow_R$ reduction is closed under substitution.
  - If $\langle \mu\alpha.c \| E \rangle \rightarrow_R \langle \mu\alpha.c \| E' \rangle$ because $E \rightarrow_R E'$ then E' must be a co-value since co-values are closed under reduction and

    $$c\{E/\alpha\} \twoheadrightarrow_R c\{E'/\alpha\} \prec_{\mu_\mathcal{S}} \langle \mu\alpha.c \| E' \rangle$$

which is decreasing because

$$\mathit{depth}(c\{E/\alpha\} \twoheadrightarrow_R c\{E'/\alpha\}) > 0 = \mathit{depth}(\langle\mu\alpha.c\|E\rangle \succ_{\mu_{\mathcal{S}}} c\{E/\alpha\})$$

- $\langle V\|\tilde{\mu}x.c\rangle \succ_{\tilde{\mu}_{\mathcal{S}}} c\{V/x\}$ is analogous to the previous case by duality.
- $\mu\alpha.\langle v\|\alpha\rangle \succ_{\eta_\mu} v$ has two different possible nested reductions:
  - If $\mu\alpha.\langle v\|\alpha\rangle \to_{\mu_{\mathcal{S}}} \mu\alpha.c\{\alpha/\beta\}$ because $v = \mu\beta.c$ then $v =_\alpha \mu\alpha.c\{\alpha/\beta\}$, so the two divergent reductions trivially join.
  - If $\mu\alpha.\langle v\|\alpha\rangle \to_R \mu\alpha.\langle v'\|\alpha\rangle$ because $v \to_R v'$ then $v' \prec_{\eta_\mu} \mu\alpha.\langle v'\|\alpha\rangle$.
- $\tilde{\mu}x.\langle x\|e\rangle \succ_{\eta_{\tilde{\mu}}} e$ is analogous to the previous case by duality.
- $\left\langle \mathsf{K}(\vec{E},\vec{V})\middle\|\tilde{\mu}[\cdots \mid \mathsf{K}(\vec{\alpha},\vec{x}).c \mid \cdots]\right\rangle \succ_{\beta_{\mathcal{S}}} c\left\{\overrightarrow{E/\alpha},\overrightarrow{V/x}\right\}$ has several possible nested reductions inside the (co-)values $\vec{E},\vec{V}$ of the data structure or inside the commands $\ldots c \ldots$ inside the case abstraction, all of which follow similarly to the latter two cases for $\mu_{\mathcal{S}}$ and $\tilde{\mu}_{\mathcal{S}}$. Otherwise, there are no other nested reductions.
- $\left\langle \mu(\cdots \mid \mathsf{O}(\vec{x},\vec{\alpha}).c \mid \cdots)\middle\|\mathsf{O}(\vec{E},\vec{V})\right\rangle \succ_{\beta_{\mathcal{S}}} c\left\{\overrightarrow{V/x},\overrightarrow{E/\alpha}\right\}$ is analogous to the previous case by duality.
- $\mathsf{K}(\vec{E},e',\vec{e},\vec{v}) \succ_{\varsigma_{\mathcal{S}}} \mu\alpha.\left\langle\mu\beta.\left\langle\mathsf{K}(\vec{E},\beta,\vec{e},\vec{v})\middle\|\alpha\right\rangle\middle\|e'\right\rangle$ has the following possible nested reductions:
  - Any reduction inside $\vec{E}$, $\vec{e}$, or $\vec{v}$ trivially joins in one step because (co-)values are closed under reduction. Likewise, any reduction inside e' which does not convert e' into a co-value also joins in one step.
  - If $\mathsf{K}(\vec{E},e',\vec{e},\vec{v}) \to_R \mathsf{K}(\vec{E},E',\vec{e},\vec{v})$ because $e' \to_R E'$ then

$$\begin{aligned}
\mu\alpha.\left\langle\mu\beta.\left\langle\mathsf{K}(\vec{E},\beta,\vec{e},\vec{v})\middle\|\alpha\right\rangle\middle\|e'\right\rangle &\to_R \mu\alpha.\left\langle\mu\beta.\left\langle\mathsf{K}(\vec{E},\beta,\vec{e},\vec{v})\middle\|\alpha\right\rangle\middle\|E'\right\rangle\\
&\to_{\mu_{\mathcal{S}}} \mu\alpha.\left\langle\mathsf{K}(\vec{E},E',\vec{e},\vec{v})\middle\|\alpha\right\rangle\\
&\succ_{\eta_\mu} \mathsf{K}(\vec{E},E',\vec{e},\vec{v})
\end{aligned}$$

    which is decreasing because the first two reductions occur in non-empty contexts (i.e. their depth is greater than 0) and the final reduction occurs in the empty context, so its measure is the same as the $\varsigma_{\mathcal{S}}$ reduction.

– All three other $\varsigma_{\mathcal{S}}$ are similar to the previous case. □

As special cases, each of the particular substitution strategies we considered in Section 5.1 (except for $\mathcal{U}$) is confluent.

Corollary 5.1. *The $\rightarrow_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}}$ reduction relation is confluent for $\mathcal{S} = \mathcal{V}$, $\mathcal{S} = \mathcal{N}$, and $\mathcal{S} = \mathcal{LV}$.*

Proof. Follows from Theorem 5.1, since each of $\mathcal{V}$, $\mathcal{N}$, and $\mathcal{LV}$ make $\succ_{\mu\tilde{\mu}}$ deterministic and their (co-)values are closed under reduction. □

Extensionality and lifting

Now that we have two competing theories for the dynamic semantics of (co-)data, how do they compare? Do they agree, and give similar results for the same programs? As it turns out, when the restriction of the $\beta\varsigma$ equational theory to typed commands and (co-)terms is derivable from the $\beta\eta$ equational theory with help from the $\mu\tilde{\mu}$ core. For example, we have the specific ς rules specialized for the $\oplus$ connective declared in Figure 5.6:

$$(\varsigma^{\oplus}) \qquad \iota_1(v) = \mu\alpha.\, \langle v \| \tilde{\mu}x.\, \langle \iota_1(x) \| \alpha \rangle\rangle \qquad (\varsigma^{\oplus}) \qquad \iota_2(v) = \mu\alpha.\, \langle v \| \tilde{\mu}x.\, \langle \iota_2(x) \| \alpha \rangle\rangle$$

These rules can be derived by η expansion followed by β reduction:

$$\begin{aligned} \iota_1(v) : A \oplus B &=_{\eta_\mu} \mu\alpha.\, \langle \iota_1(v) \| \alpha \rangle \\ &=_{\eta^{\oplus}} \mu\alpha.\, \langle \iota_1(v) \| \tilde{\mu}[\iota_1(x).\langle \iota_1(x) \| \alpha \rangle \mid \ldots] \rangle \\ &=_{\beta^{\oplus}} \mu\alpha.\, \langle v \| \tilde{\mu}x.\, \langle \iota_1(x) \| \alpha \rangle\rangle \end{aligned}$$

Notice here that the steps of this derivation are captured exactly by our formulation of β and η axioms: (1) the ability to η expand a co-variable, and (2) the ability to perform β reduction immediately to break apart a structure once the constructor is seen.

We also have similar specialized ς rules for functions:

$$(\varsigma^{\rightarrow}) \qquad v \cdot e = \tilde{\mu}x.\, \langle v \| \tilde{\mu}y.\, \langle x \| y \cdot e \rangle\rangle \qquad (\varsigma^{\rightarrow}) \qquad V \cdot e = \tilde{\mu}x.\, \langle \mu\alpha.\, \langle x \| V \cdot \alpha \rangle \| e \rangle$$

which are again derivable by a similar procedure of η expansion and β reduction:

$$\begin{aligned}
V \cdot e : A \to B &=_{\eta_{\tilde{\mu}}} \tilde{\mu}x.\, \langle x \| V \cdot e \rangle \\
&=_{\eta^{\to}} \tilde{\mu}x.\, \langle \mu(y \cdot \alpha.\langle x \| y \cdot \alpha \rangle) \| V \cdot e \rangle \\
&=_{\beta^{\to}} \tilde{\mu}x.\, \langle V \| \tilde{\mu}y.\, \langle \mu\alpha.\, \langle x \| y \cdot \alpha \rangle \| e \rangle \rangle \\
&=_{\tilde{\mu}_V} \tilde{\mu}x.\, \langle \mu\alpha.\, \langle x \| V \cdot \alpha \rangle \| e \rangle
\end{aligned}$$

$$\begin{aligned}
v \cdot e : A \to B &=_{\eta_{\tilde{\mu}}} \tilde{\mu}x.\, \langle x \| v \cdot e \rangle \\
&=_{\eta^{\to}} \tilde{\mu}x.\, \langle \mu(y \cdot \alpha.\langle x \| y \cdot \alpha \rangle) \| v \cdot e \rangle \\
&=_{\beta^{\to}} \tilde{\mu}x.\, \langle v \| \tilde{\mu}y.\, \langle \mu\alpha.\, \langle x \| y \cdot \alpha \rangle \| e \rangle \rangle \\
&=_{\tilde{\mu}_V} \tilde{\mu}x.\, \langle v \| \tilde{\mu}y.\, \langle x \| \tilde{\mu}x.\, \langle \mu\alpha.\, \langle x \| y \cdot \alpha \rangle \| e \rangle \rangle \rangle \\
&=_{\varsigma^{\to}} \tilde{\mu}x.\, \langle v \| \tilde{\mu}y.\, \langle x \| y \cdot e \rangle \rangle
\end{aligned}$$

These particular ς axioms for functions are interesting because they were left out of Wadler's (2003) sequent calculus, however, we now know they were implicitly present in the equational theory (Wadler, 2005) as a consequence of the β and η axioms. This same procedure words for all the definable (co-)data types, so that the $\beta\eta$ axioms for the F and G (co-)data type constructors as declared in Figure 5.8 generate the derived ς axioms shown in Figure 5.12. These rules search for the left-most non-value or non-co-value found in a data or co-data structure, and give it a name with an input or output abstraction, which comes from the ordering of bindings implied by the β laws in Figure 5.11. For example, the instance of the derived lift axioms for pair types $A \otimes B$, following the general pattern, are:

$$(\varsigma^{\otimes}) \quad (v, v') = \mu\alpha.\, \langle v \| \tilde{\mu}x.\, \langle (x, v') \| \alpha \rangle \rangle \qquad (\varsigma^{\otimes}) \quad (V, v') = \mu\alpha.\, \langle v' \| \mu y.\, \langle (V, y) \| \alpha \rangle \rangle$$

Remark 5.5. Notice that all of the strategies we have considered so far follow a particular pattern. More specifically, each of the $\mathcal{V}$, $\mathcal{N}$, and $\mathcal{LV}$ strategies fit the following *focalizing* criteria.

Definition 5.2 (Focalizing strategy). A strategy $\mathcal{S}$ is *focalizing* if and only if

- (co-)variables are (co-)values (as assumed to hold for all strategies),
- structures built from (co-)values are themselves (co-)values (i.e. $\mathsf{K}(\vec{E}, \vec{V})$ and $\mathsf{O}[\vec{V}, \vec{E}]$ are (co-)values), and

- case abstractions are (co-)values (i.e. $\tilde{\mu}[\mathsf{K}(\vec{\alpha},\vec{x}).c \mid \ldots]$ and $\mu(\mathsf{O}[\vec{x},\vec{\alpha}].c \mid \ldots)$ are (co-)values).

These criteria correspond to the impact of focalization on the typing rules for system L from Section 4.4, and further justifies the connection between maintaining focus with the stoup in proof search and values and strictness in languages. Furthermore, it also happens that the non-(co-)values of each of these three strategies are closed under ς-reduction as well. In other words, the ς laws cannot create or destroy (co-)values, but instead only serve to identify and lift out sub-(co-)terms that are out of focus. Thus, these strategies are all *focalizing*, in that they follow a focalization procedure dynamically at run-time.

Besides demonstrating the connection between focalization and evaluation, these criteria give us a general technique for developing strategies. In particular, we can take a core strategy, which covers only the structural core of the sequent calculus, and automatically extend it with data and co-data with a single, generic method. First, close the sets of values and co-values under the above three focalization criteria, so that $\mathsf{K}(\vec{E},\vec{V})$, $\mathsf{O}[\vec{V},\vec{E}]$, $\tilde{\mu}[\mathsf{K}(\vec{\alpha},\vec{x}).c \mid \ldots]$, and $\mu(\mathsf{O}[\vec{x},\vec{\alpha}].c \mid \ldots)$ are all (co-)values. Second, close the sets of values and co-values under ς expansion, so that if $v \to_\varsigma V$ and $e \to_\varsigma E$ then v and e are themselves (co-)values.

This generic method let's us generate the previously known strategies for the parametric sequent calculus. For example, applying this method to the core $\mathcal{V}$, $\mathcal{N}$, and $\mathcal{LV}$ strategies from Figures 5.2 and 5.3 gives exactly the extended strategies in Figures 5.9 and 5.10. So the core strategy gives enough information to recover its corresponding focalizing strategy. Furthermore, we already assumed that strategies always consider (co-)variables to be (co-)values. Thus, in the world of focalizing strategies for the parametric sequent calculus, the only crucial decision is what to do with general input and output abstractions; everything else follows from focalization. *End remark* 5.5.

More generally, we can say that the $\beta\varsigma$ equational theory of (co-)data is sound with respect to the $\beta\eta$ equational theory, with help from the core $\mu\tilde{\mu}$ theory of substitution.

Theorem 5.2 (Soundness of $\beta\varsigma$ w.r.t. $\beta\eta$)**.** *For any substitution strategy $\mathcal{S}$:*

a) If $c : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$, $c' : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$, and $c =_{\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}} c'$, then $c =_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} c'$.

b) If $\Gamma \vdash_{\mathcal{G}} v : A|\Delta$, $\Gamma \vdash_{\mathcal{G}} v' : A|\Delta$, and $v =_{\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}} v'$, then $v =_{\beta^{\mathcal{G}}\eta^{\mathcal{G}}} v'$.

c) If $\Gamma | e : A \vdash_{\mathcal{G}} \Delta$, $\Gamma | e' : A \vdash_{\mathcal{G}} \Delta$, *and* $e =_{\beta_{\mathcal{S}} \varsigma_{\mathcal{S}}} e'$, *then* $e =_{\mu_{\mathcal{S}} \tilde{\mu}_{\mathcal{S}} \eta_\mu \eta_{\tilde{\mu}} \beta^{\mathcal{G}} \eta^{\mathcal{G}}} e'$.

Proof. Note that compatibility, reflexivity, symmetry, and transitivity of $=_{\beta_{\mathcal{S}} \varsigma_{\mathcal{S}}}$ implies the same in $=_{\mu_{\mathcal{S}} \tilde{\mu}_{\mathcal{S}} \eta_\mu \eta_{\tilde{\mu}} \beta^{\mathcal{G}} \eta^{\mathcal{G}}}$, so we only need to check that the $\succ_{\beta_{\mathcal{S}} \varsigma_{\mathcal{S}}}$ rewriting rules can be derived as $=_{\mu_{\mathcal{S}} \tilde{\mu}_{\mathcal{S}} \eta_\mu \eta_{\tilde{\mu}} \beta^{\mathcal{G}} \eta^{\mathcal{G}}}$ equalities:

- $\beta_{\mathcal{S}}$ restricted to a data type $\mathsf{F}(\vec{C})$ is derived as:

$$\begin{aligned}\left\langle \mathsf{K}(\vec{E}, \vec{V}) \middle\| \tilde{\mu}[\cdots \mid \mathsf{K}(\vec{\alpha}, \vec{x}).c \mid \cdots] \right\rangle &=_{\beta^{\mathsf{F}}} \left\langle \mu\vec{\alpha}. \left\langle \vec{V} \middle\| \tilde{\mu}\vec{x}.c \right\rangle \middle\| \vec{E} \right\rangle \\ &=_{\mu_{\mathcal{S}} \tilde{\mu}_{\mathcal{S}}} c \left\{ \overrightarrow{E/\alpha}, \overrightarrow{V/x} \right\}\end{aligned}$$

- $\beta_{\mathcal{S}}$ restricted to a co-data type $\mathsf{G}(\vec{C})$ is derived analogously to the previous case.

- $\varsigma_{\mathcal{S}}$ restricted to a data type $\mathsf{F}(\vec{C})$ is derived inductively on the structure of constructions from right-to-left as:

$$\begin{aligned}&\mathsf{K}(\vec{E}, \vec{V}, v', \vec{v}) : \mathsf{F}(\vec{C}) \\ &=_{\eta_\mu} \mu\alpha. \left\langle \mathsf{K}(\vec{E}, \vec{V}, v', \vec{v}) \middle\| \alpha \right\rangle \\ &=_{\eta^{\mathsf{F}}} \mu\alpha. \left\langle \mathsf{K}(\vec{E}, \vec{V}, v', \vec{v}) \middle\| \tilde{\mu}\left[\cdots \mid \mathsf{K}(\vec{\beta}, \vec{x}, y, \vec{z}).\left\langle \mathsf{K}(\vec{\beta}, \vec{x}, y, \vec{z}) \middle\| \alpha \right\rangle \mid \cdots\right] \right\rangle \\ &=_{\beta^{\mathsf{F}}} \mu\alpha. \left\langle \mu\vec{\beta}. \left\langle \vec{V} \middle\| \tilde{\mu}\vec{x}. \left\langle v' \middle\| \tilde{\mu}y. \left\langle \vec{v} \middle\| \tilde{\mu}z. \left\langle \mathsf{K}(\vec{\beta}, \vec{x}, y, \vec{z}) \middle\| \alpha \right\rangle \right\rangle \right\rangle \right\rangle \middle\| \vec{E} \right\rangle \\ &=_{\mu_{\mathcal{S}}} \mu\alpha. \left\langle \vec{V} \middle\| \tilde{\mu}\vec{x}. \left\langle v' \middle\| \tilde{\mu}y. \left\langle \vec{v} \middle\| \tilde{\mu}z. \left\langle \mathsf{K}(\vec{E}, \vec{x}, y, \vec{z}) \middle\| \alpha \right\rangle \right\rangle \right\rangle \right\rangle \\ &=_{\tilde{\mu}_{\mathcal{S}}} \mu\alpha. \left\langle v' \middle\| \tilde{\mu}y. \left\langle \vec{v} \middle\| \tilde{\mu}z. \left\langle \mathsf{K}(\vec{E}, \vec{V}, y, \vec{z}) \middle\| \alpha \right\rangle \right\rangle \right\rangle \\ &=_{\varsigma^{\mathsf{F}}} \mu\alpha. \left\langle v' \middle\| \tilde{\mu}y. \left\langle \mathsf{K}(\vec{E}, \vec{V}, y, \vec{v}) \middle\| \alpha \right\rangle \right\rangle\end{aligned}$$

$$\begin{aligned}&\mathsf{K}(\vec{E}, e', \vec{e}, \vec{v}) : \mathsf{F}(\vec{C}) \\ &=_{\eta_\mu} \mu\alpha. \left\langle \mathsf{K}(\vec{E}, e', \vec{e}, \vec{v}) \middle\| \alpha \right\rangle \\ &=_{\eta^{\mathsf{F}}} \mu\alpha. \left\langle \mathsf{K}(\vec{E}, e', \vec{e}, \vec{v}) \middle\| \tilde{\mu}\left[\cdots \mid \mathsf{K}(\vec{\beta}, \gamma, \vec{\delta}, \vec{x}).\left\langle \mathsf{K}(\vec{\beta}, \gamma, \vec{\delta}, \vec{x}) \middle\| \alpha \right\rangle \mid \cdots\right] \right\rangle \\ &=_{\beta^{\mathsf{F}}} \mu\alpha. \left\langle \mu\vec{\beta}. \left\langle \mu\gamma. \left\langle \mu\vec{\delta}. \left\langle \vec{v} \middle\| \tilde{\mu}\vec{x}. \left\langle \mathsf{K}(\vec{\beta}, \gamma, \vec{\delta}, \vec{x}) \middle\| \alpha \right\rangle \right\rangle \middle\| \vec{e} \right\rangle \middle\| e' \right\rangle \middle\| \vec{E} \right\rangle \\ &=_{\mu_{\mathcal{S}}} \mu\alpha. \left\langle \mu\gamma. \left\langle \mu\vec{\delta}. \left\langle \vec{v} \middle\| \tilde{\mu}\vec{x}. \left\langle \mathsf{K}(\vec{E}, \gamma, \vec{\delta}, \vec{x}) \middle\| \alpha \right\rangle \right\rangle \middle\| \vec{e} \right\rangle \middle\| e' \right\rangle \\ &=_{\varsigma^{\mathsf{F}}} \mu\alpha. \left\langle \mu\gamma. \left\langle \mathsf{K}(\vec{E}, \gamma, \vec{e}, \vec{v}) \middle\| \alpha \right\rangle \middle\| e' \right\rangle\end{aligned}$$

- $\varsigma_{\mathcal{S}}$ restricted to a co-data type $\mathsf{G}(\vec{C})$ is derived analogously to the previous case. □

Going the other way, the strategy-independent β law is sound with respect to the strategy-dependent $\beta\varsigma$ rewriting theory, with the help from the core $\mu\tilde{\mu}_{\mathcal{S}}$ theory of substitution, for any *focalizing* strategy (Definition 5.2).

Theorem 5.3 (Soundness of β w.r.t. $\beta\varsigma$)**.** *For any focalizing strategy $\mathcal{S}$, if $c =_{\beta^{\mathcal{G}}} c'$, then $c =_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}} c'$, and similarly for (co-)terms.*

Proof. Note that compatibility, reflexivity, symmetry, and transitivity of $=_{\beta^{\mathcal{G}}}$ implies the same in $=_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}}$, so we only need to check that the $\succ_{\beta^{\mathcal{G}}}$ rewriting rules can be derived as $=_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}}$ equalities:

– $\beta_{\mathcal{S}}$ for a data structure is derived as:

$$\begin{aligned}
&\langle \mathsf{K}(\vec{e}, \vec{v}) \| \tilde{\mu}[\cdots \mid \mathsf{K}(\vec{\alpha}, \vec{x}).c \mid \cdots] \rangle \\
&=_{\varsigma_{\mathcal{S}}\mu_{\mathcal{S}}} \langle \mu\vec{\alpha}.\, \langle \mathsf{K}(\vec{\alpha}, \vec{v}) \| \tilde{\mu}[\cdots \mid \mathsf{K}(\vec{\alpha}, \vec{x}).c \mid \cdots] \rangle \| \vec{e} \rangle \\
&=_{\varsigma_{\mathcal{S}}\mu_{\mathcal{S}}\mu_{\mathcal{S}}} \langle \mu\vec{\alpha}.\, \langle \vec{v} \| \tilde{\mu}\vec{x}.\, \langle \mathsf{K}(\vec{\alpha}, \vec{x}) \| \tilde{\mu}[\cdots \mid \mathsf{K}(\vec{\alpha}, \vec{x}).c \mid \cdots] \rangle \rangle \| \vec{e} \rangle \\
&=_{\beta_{\mathcal{S}}} \langle \mu\vec{\alpha}.\, \langle \vec{v} \| \tilde{\mu}\vec{x}.c \rangle \| \vec{e} \rangle
\end{aligned}$$

The first two steps follow by applying $\varsigma_{\mathcal{S}}$ reduction to name non-(co-)values and applying $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}$ to name (co-)values, and then substituting the case abstraction (which must be a co-value because $\mathcal{S}$ is focalizing) for the outer μ-abstraction generated by $\varsigma_{\mathcal{S}}$. The last step follows because (co-)variables are (co-)values since $\mathcal{S}$ is focalizing.

– $\beta_{\mathcal{S}}$ for a co-data structure is derived analogously to the previous case. □

So equationally speaking, in the presence of the core $\mu\tilde{\mu}$ theory of substitution, typed versions of the $\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}$ laws can be derived from the typed $\beta^{\mathcal{G}}\eta^{\mathcal{G}}$ laws, and untyped versions of the $\beta^{\mathcal{G}}$ laws can be derived from the untyped $\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}$ laws. However, the typed $\eta^{\mathcal{G}}$ law cannot be derived by $\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}$, so $\beta^{\mathcal{G}}\eta^{\mathcal{G}}$ equates more typed programs.

Combining Strategies in Connectives

The parametric $\mu\tilde{\mu}$-calculus provides a general framework for describing all the basic connectives discussed in Section 5.2, giving a mechanism for extending the syntax and semantics of the sequent calculus to account for a wide variety of new structures. However, what about the connectives of polarized system L from Section 4.3 which involved both polarities? Can we include the shifts, negation, and polarized function type into our notion of user-defined data and (co-)data types? Also, what about polarized logic ability to utilize multiple evaluation strategies in a single program? Is there a way to instantiate the parametric equational theory with two strategies at the same time? Or even more than two strategies at once?

To answer to all of these questions, let's look at how the parametric $\mu\tilde{\mu}$-calculus described thus far compares to a polarized languages like system L. In polarized system L, all types are classified by one of two polarities: positive or negative. The distinction between data and co-data determines the polarity of a type, and furthermore the type's polarity determines the evaluation order used for programs of that type. In polarized system L, data types are positive and describe call-by-value programs, whereas co-data types are negative and describe call-by-name programs. In the parametric $\mu\tilde{\mu}$-calculus, we have stepped outside this regimine, so that programs of data types and co-data types can be evaluated with the strategy of our own choosing. However, we can still allow for this choice of strategy while remaining compatible with polarized logic's type-based approach to evaluation strategy. In particular, we can still have multiple classifications of types, as a generalization of polarized types, and use the type's classification to determine which strategy to use for programs of that type. In other words, even though we have decoupled the link between data vs co-data and evaluation order, we can still have the evaluation strategy depend on the type.

Separating types into different classifications is not a new idea, and shows up in several type systems in the form of *kinds*. Effectively, kinds classify types in the same way that types classify terms, i.e. kinds are types "one level up the chain." Therefore, we will look at extending the parametric $\mu\tilde{\mu}$-calculus with multiple base kinds for classifying (co-)data types of different strategies. For example, if we are interested in both call-by-value ($\mathcal{V}$) and call-by-name ($\mathcal{N}$) evaluation, then we would have two different base kinds, called $\mathcal{V}$ and $\mathcal{N}$, which classify the various types of call-by-value and call-by-name programs, respectively.[4] This extension to the language of kinds involves understanding more about which kinds involved in the various connectives: we to know the kinds of types expected as parameters to the connective, as well as

[4] Here we use the names $\mathcal{V}$ and $\mathcal{N}$ to mean both a strategy (a set of values, co-values, and evaluation contexts) and a kind (a "type of types"). Even though the two are different things, the clash in naming is meant to make obvious the connection between the kind and the strategy. Both kinds and strategies are used in very different places, so the meaning of $\mathcal{V}$ and $\mathcal{N}$ can be distinguished from context.

the kind of type the connective builds. Thus, we need to be more explicit in our data and co-data declarations in order to specify the link with strategy.

For example, let's suppose we want a wholly call-by-value pair type, corresponding to the polarized version of the positive $\otimes$ connective. We can make this intent known by adding explicit kind annotations to the declaration of $\otimes$ from Figure 5.6:[5]

$$\begin{aligned}&\mathbf{data}\,(X : \mathcal{V}) \otimes (Y : \mathcal{V}) : \mathcal{V}\,\mathbf{where}\\ &\quad (\_,\_) : X : \mathcal{V}, Y : \mathcal{V} \vdash X \otimes Y : \mathcal{V} \mid\end{aligned}$$

Here, we say that the types for both components of the pair belong to kind $\mathcal{V}$, and the resulting pair type itself also belongs to kind $\mathcal{V}$. Because we interpret the kind $\mathcal{V}$ as containing the types of programs which should be evaluated according to the $\mathcal{V}$ strategy, then this declaration gives us the basic pair type in the call-by-value instance of the parametric equational theory. The main difference here is that we are being explicit about the fact that types A, B, and $A \otimes B$ must be call-by-value, and cannot be interpreted by any other evaluation strategy, as opposed to the previous situation where the programs of a (co-)data type could be interpreted by any evaluation strategy of our choice. The impact of these explicit kind annotation on typing is minor: the rules for typing terms and co-terms of type $A \otimes B$ are essentially the same as before. The main change is that we need to make sure that types are well-kinded. In particular, we have a new judgement $X_1 : k_1, \dots, X_n : k_n \vdash_{\mathcal{G}} A : k$ that says that A is a type of kind k with respect to the assigned kinds of type variables in the typing environment $\Theta = X_1 : k_1, \dots, X_n : k_n$ and the declarations in $\mathcal{G}$. Then $A \otimes B$ is a type of kind $\mathcal{V}$ under an typing environment Θ and set of declarations $\mathcal{G}$ containing the data declaration of $\otimes$ when both A and B are as well:

$$\frac{\Theta \vdash_{\mathcal{G}} A : \mathcal{V} \quad \Theta \vdash_{\mathcal{G}} A : \mathcal{V}}{\Theta \vdash_{\mathcal{G}} A \otimes B : \mathcal{V}}$$

Additionally, we can also describe a wholly call-by-name product type, corresponding to the polarized version of the negative & connective. Making this intent known in the more general setting is done by adding $\mathcal{N}$ kind annotations to

[5] Adding explicit kinds to a data type declaration is not new; it is supported by GHC with the extension "kind signatures." Rather, the new idea is to have the kind impact the meaning of a term by denoting its evaluation strategy.

$$\begin{array}{ll}
\mathbf{data}\,(X:\mathcal{V})\oplus(Y:\mathcal{V}):\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,(X:\mathcal{N})\mathbin{\&}(Y:\mathcal{N}):\mathcal{N}\,\mathbf{where} \\
\quad \iota_1 : X:\mathcal{V}\vdash X\oplus Y:\mathcal{V}\mid & \quad \pi_1 : \;\mid X\mathbin{\&}Y:\mathcal{N}\vdash X:\mathcal{N} \\
\quad \iota_2 : Y:\mathcal{V}\vdash X\oplus Y:\mathcal{V}\mid & \quad \pi_2 : \;\mid X\mathbin{\&}Y:\mathcal{N}\vdash Y:\mathcal{N} \\
\\
\mathbf{data}\,(X:\mathcal{V})\otimes(Y:\mathcal{V}):\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,(X:\mathcal{N})\parr(Y:\mathcal{N}):\mathcal{N}\,\mathbf{where} \\
\quad (\_,\_) : X:\mathcal{V},Y:\mathcal{V}\vdash X\otimes Y:\mathcal{V}\mid & \quad [\_,\_] : \;\mid X\parr Y:\mathcal{N}\vdash X:\mathcal{N},Y:\mathcal{N} \\
\\
\mathbf{data}\,1:\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,\bot:\mathcal{N}\,\mathbf{where} \\
\quad () : \;\vdash 1:\mathcal{V}\mid & \quad [] : \;\mid\bot:\mathcal{N}\vdash \\
\\
\mathbf{data}\,0:\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,\top:\mathcal{N}\,\mathbf{where}
\end{array}$$

FIGURE 5.13. Declarations of the basic single-strategy data and co-data types.

the declaration of & from Figure 5.6:

$$\begin{array}{l}
\mathbf{codata}\,(X:\mathcal{N})\mathbin{\&}(Y:\mathcal{N}):\mathcal{N}\,\mathbf{where} \\
\quad \pi_1 : \;\mid X\mathbin{\&}Y:\mathcal{N}\vdash X:\mathcal{N} \\
\quad \pi_2 : \;\mid X\mathbin{\&}Y:\mathcal{N}\vdash Y:\mathcal{N}
\end{array}$$

Here, we say that the types for both components of the product belong to the kind $\mathcal{N}$, and the resulting product type itself also belongs to kind $\mathcal{N}$. Thus, this declaration forces us to evaluate programs of this type in a way that matches the corresponding interpretation in polarized languages like system L. In general, we can annotate all the basic types of Figure 5.6 to force them into their polarized interpretations, giving the annotated declarations in Figure 5.13. Essentially, this process involves us annotating all data types with the kind $\mathcal{V}$ and all co-data types with the kind $\mathcal{N}$, following the assertion that data types describe call-by-value evaluation and co-data types describe call-by-name evaluation. As before, the typing rules for terms and co-terms of type $A \mathbin{\&} B$ do not change with the addition of kind annotations, we only have an additional rule for the well-kinded uses of the & connective:

$$\frac{\Theta\vdash_{\mathcal{G}} A:\mathcal{N} \quad \Theta\vdash_{\mathcal{G}} B:\mathcal{N}}{\Theta\vdash_{\mathcal{G}} A\mathbin{\&}B:\mathcal{N}}$$

$$
\begin{array}{ll}
\mathbf{data}\, {\downarrow}(X : \mathcal{N}) : \mathcal{V}\, \mathbf{where} & \mathbf{codata}\, {\uparrow}(X : \mathcal{V}) : \mathcal{N}\, \mathbf{where} \\
\quad {\downarrow} : X : \mathcal{N} \vdash {\downarrow}X : \mathcal{V} \mid & \quad {\uparrow} : \ \mid {\uparrow}X : \mathcal{N} \vdash X : \mathcal{V} \\
 & \\
\mathbf{data}\, {\sim}(X : \mathcal{N}) : \mathcal{V}\, \mathbf{where} & \mathbf{codata}\, \neg(X : \mathcal{V}) : \mathcal{N}\, \mathbf{where} \\
\quad {\sim} : \ \vdash {\sim}X : \mathcal{V} \mid X : \mathcal{N} & \quad \neg : X : \mathcal{V} \mid \neg X : \mathcal{N} \vdash \\
 & \\
\mathbf{data}\, (X : \mathcal{V}) - (Y : \mathcal{N}) : \mathcal{V}\, \mathbf{where} & \mathbf{codata}\, (X : \mathcal{V}) \to (Y : \mathcal{N}) : \mathcal{N}\, \mathbf{where} \\
\quad \_ \cdot \_ : X : \mathcal{V} \vdash (X - Y) : \mathcal{V} \mid Y : \mathcal{N} & \quad \_ \cdot \_ : X : \mathcal{V} \mid (X \to Y) : \mathcal{N} \vdash Y : \mathcal{N}
\end{array}
$$

FIGURE 5.14. Declarations of basic mixed-strategy data and co-data types.

While annotating the kinds of types involved in (co-)data declarations is relatively straightforward for the single-polarity connectives, the exercise becomes more important when representing polarized connectives that involve both polarities. For example, the polarized function type made non-trivial use of both polarities in its definition, which can be captured by the following annotated co-data declaration:

$$
\begin{array}{l}
\mathbf{codata}\, (X : \mathcal{V}) \to (Y : \mathcal{N}) : \mathcal{N}\, \mathbf{where} \\
\quad \_ \cdot \_ : \quad X : \mathcal{V} \mid X \to Y : \mathcal{N} \vdash Y : \mathcal{N}
\end{array}
$$

Intuitively, the source A of the function type must be positive so it belongs to the kind $\mathcal{V}$, and the target B of the function type must be negative so it belongs to kind $\mathcal{N}$. Furthermore, since polarized languages assume that all co-data types themselves are negative, the overall type $A \to B$ belongs to the kind $\mathcal{N}$. This declaration gives us the primordial, Zeilberger's (2009) polarized function type, with the same impact on evaluation order. Likewise, we can give annotated (co-)data type declarations for other mixed-polarity connectives, like the polarity shifts ${\downarrow}A$ and ${\uparrow}A$ and involutive negations $\neg A$ and ${\sim}A$, as shown in Figure 5.14. Thus, kind-annotated (co-)data type declarations give us a syntactic mechanism for summarizing all the simple polarized connectives that we have previously seen.

In general, the extension of (co-)data declarations to include multiple base kinds ($\mathcal{R}$, $\mathcal{S}$, $\mathcal{T}$), along with the necessary kinding restrictions, is given in Figure 5.15. This extension means that we need to keep track of what kind each type variable has, since

[6] This is just shorthand for a (co-)data declaration of $\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}$ in $\mathcal{G}$.

$$k \in \mathit{Kind} ::= \mathcal{S} \qquad\qquad \mathcal{R}, \mathcal{S}, \mathcal{T} \in \mathit{BaseKind} ::= \dots$$

$$A, B, C \in \mathit{Type} ::= X \mid \mathsf{F}(\overrightarrow{A}) \quad X, Y, Z \in \mathit{TypeVariable} ::= \dots \quad \mathsf{F}, \mathsf{G} \in \mathit{TypeCon} ::= \dots$$

$$\begin{aligned} \mathit{decl} \in \mathit{Declaration} ::={}& \mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B : \mathcal{R}}\right)} \\ \mid{}& \mathbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(\overrightarrow{X}) \vdash \overrightarrow{B : \mathcal{R}}\right)} \end{aligned}$$

$$\mathcal{G} \in \mathit{GlobalEnv} ::= \overrightarrow{\mathit{decl}} \qquad\qquad \Theta \in \mathit{TypeEnv} ::= \overrightarrow{X : k}$$

$$\Gamma \in \mathit{InputEnv} ::= \overrightarrow{x : A} \qquad\qquad \Delta \in \mathit{OutputEnv} ::= \overrightarrow{\alpha : A}$$

$$J, H \in \mathit{Judgement} ::= (\mathcal{G} \vdash \mathit{decl}) \mid (\Theta \vdash_{\mathcal{G}} A : k)$$

Declaration rules:

$$\frac{\overrightarrow{\overrightarrow{X : k} \vdash_{\mathcal{G}} A : \mathcal{T}} \quad \overrightarrow{\overrightarrow{X : k} \vdash_{\mathcal{G}} B : \mathcal{R}}}{\mathcal{G} \vdash \begin{array}{c} \mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where} \\ \overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B : \mathcal{R}}\right)} \end{array}}\ \mathit{data} \qquad \frac{\overrightarrow{\overrightarrow{X : k} \vdash_{\mathcal{G}} A : \mathcal{T}} \quad \overrightarrow{\overrightarrow{X : k} \vdash_{\mathcal{G}} B : \mathcal{R}}}{\mathcal{G} \vdash \begin{array}{c} \mathbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where} \\ \overrightarrow{\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{F}(\overrightarrow{X}) \vdash \overrightarrow{B : \mathcal{R}}\right)} \end{array}}\ \mathit{codata}$$

Kind rules:

$$\frac{}{\Theta, X : k \vdash_{\mathcal{G}} X : k}\ \mathit{TV} \qquad\qquad \frac{\overrightarrow{\Theta \vdash_{\mathcal{G}} C : k} \quad (\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S})^{6} \in \mathcal{G}}{\Theta \vdash_{\mathcal{G}} \mathsf{F}(\overrightarrow{C}) : \mathcal{S}}\ \mathsf{F}T$$

FIGURE 5.15. Kinds of multi-strategy (co-)data declarations and types.

there are now multiple options, necessitating the introduction of type environments $X_1 : k_1, \dots, X_2 : k_2$ denoted by Θ which are analogous to input (Γ) and output (Δ) environments at the level of types instead of programs. These type environments Θ are used for checking the kind of a type A as in the first new form of judgement $\Theta \vdash_{\mathcal{G}} A : k$ which checks that the type A has kind k under the assumption that type variables have the kind listed in Θ given the set of declarations $\mathcal{G}$. Since all the specific types are generated by (co-)data declarations, there are only two inference rules for finding the kind of a type: reference to a type variable (TV) or an instance of a particular (co-)data type former $\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}$ from the global set of declarations $\mathcal{G}$. We annotate the types and type variables in (co-)data declarations to make the intension of the declaration explicit in the syntax. This explicit annotation makes it straightforward to check that declarations are well-formed. The second new form of judgement $\mathcal{G} \vdash \mathit{decl}$ checks that the declaration decl is well-formed—meaning that it includes only well-kinded types—given a previously established set of declarations $\mathcal{G}$.

To accomodate the generalization to multiple base kinds, we must also update the typing rules for programs of the parametric $\mu\tilde{\mu}$-calculus, as shown in Figure 5.16. For the most part, the change from the single-kinded type system from Figure 5.8 is that we thread the type environment Θ around the rules, as demonstrated by the updated judgement forms $c : \left(\Gamma \vdash_{\mathcal{G}}^{\Theta} \Delta\right)$, $\Gamma \vdash_{\mathcal{G}}^{\Theta} v : A \mid \Delta$, and $\Gamma \mid e : A \vdash_{\mathcal{G}}^{\Theta} \Delta$. Note that the only substantial update in the typing rules is in the cut rule: *Cut* now takes an additional premise $\Theta \vdash_{\mathcal{G}} A : \mathcal{S}$ checking that the cut type is indeed a type of some base kind $\mathcal{S}$. This extra premise is needed because, reading the rules bottom-up, the *Cut* is the only inference rule that invents a new type out of thin air (see Section 3.1). It is therefore prudent to check that this new type actually makes sense under the given type environment Θ and global declarations $\mathcal{G}$. Other than this change to *Cut*, the other core inference rules (*VR*, *VL*, *AR*, *AL*) and the logical rules are essentially the same as from Figure 5.8, ignoring Θ.

Having outlined the general pattern for mixed-strategy (co-)data types, we can use the declaration mechanism to come up with special-purpose types that might be used in a program. For example, we can represent the use of strictness in Haskell to create lazy data structures with strict fields, like a lazy pair where the first component is strict. We can signify this intent by declaring a different pair type that uses two

$$\mathit{Judgement} ::= c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right) \mid (\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta)$$

Core rules:

$$\frac{}{x : A \vdash^{\Theta}_{\mathcal{G}} x : A \mid}\;\mathit{VR} \qquad \frac{}{\mid \alpha : A \vdash^{\Theta}_{\mathcal{G}} \alpha : A}\;\mathit{VL}$$

$$\frac{c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\alpha.c : A \mid \Delta}\;\mathit{AR} \qquad \frac{c : \left(\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}x.c : A \vdash^{\Theta}_{\mathcal{G}} \Delta}\;\mathit{AL}$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta \quad \Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \Gamma' \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta'}{\langle v \| e \rangle : \left(\Gamma', \Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta', \Delta\right)}\;\mathit{Cut}$$

Logical rules:

Given $\mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{\Gamma'_j \mid e : B_{ij}\{\overrightarrow{C/X}\} \vdash^{\Theta}_{\mathcal{G}} \Delta'_j}^{j} \quad \overrightarrow{\Gamma_j \mid v : A_{ij}\{\overrightarrow{C/X}\} \vdash^{\Theta}_{\mathcal{G}} \Delta_j}^{j}}{\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \vdash^{\Theta}_{\mathcal{G}} \mathsf{K}_i(\overrightarrow{e}, \overrightarrow{v}) : \mathsf{F}(\overrightarrow{C}) \mid \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}}\;\mathsf{F}R_{\mathsf{K}_i}$$

$$\frac{\overrightarrow{c_i : \left(\Gamma, \overrightarrow{x_i : A_i\{\overrightarrow{C/X}\}} \vdash^{\Theta}_{\mathcal{G}} \overrightarrow{\alpha_i : B_i\{\overrightarrow{C/X}\}}, \Delta\right)}^{i}}{\Gamma \mid \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_i}, \overrightarrow{x_i}).c_i}^{i}\right] : \mathsf{F}(\overrightarrow{C}) \vdash^{\Theta}_{\mathcal{G}} \Delta}\;\mathsf{F}L$$

Given $\mathbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\overrightarrow{X}) \vdash \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{c_i : \left(\Gamma, \overrightarrow{x_i : A_i\{\overrightarrow{C/X}\}} \vdash^{\Theta}_{\mathcal{G}} \overrightarrow{\alpha_i : B_i\{\overrightarrow{C/X}\}}, \Delta\right)}^{i}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\left(\overrightarrow{\mathsf{O}_i[\overrightarrow{x_i}, \overrightarrow{\alpha_i}].c_i}^{i}\right) : \mathsf{G}(\overrightarrow{C}) \mid \Delta}\;\mathsf{G}R$$

$$\frac{\overrightarrow{\Gamma_j \mid v : A_{ij}\{\overrightarrow{C/X}\} \vdash^{\Theta}_{\mathcal{G}} \Delta_j}^{j} \quad \overrightarrow{\Gamma'_j \mid e : B_{ij}\{\overrightarrow{C/X}\} \vdash^{\Theta}_{\mathcal{G}} \Delta'_j}^{j}}{\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \mid \mathsf{O}_i[\overrightarrow{v}, \overrightarrow{e}] : \mathsf{G}(\overrightarrow{C}) \vdash^{\Theta}_{\mathcal{G}} \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}}\;\mathsf{G}L_{\mathsf{O}_i}$$

FIGURE 5.16. Types of multi-strategy (co-)data in the parametric $\mu\tilde{\mu}$ sequent calculus.

different kinds, $\mathcal{N}$ and $\mathcal{V}$:

$$\begin{aligned}&\mathbf{data}\,\mathsf{MixedPair}(X : \mathcal{V}, Y : \mathcal{N}) : \mathcal{N}\,\mathbf{where}\\ &\qquad \mathsf{MPair} : X : \mathcal{V}, Y : \mathcal{N} \vdash \mathsf{MixedPair}(X, Y) : \mathcal{N} \mid\end{aligned}$$

In this declaration, the fact that the type A belongs to kind $\mathcal{V}$ denotes that the first component should be evaluated with the call-by-value strategy $\mathcal{V}$, whereas the second component and the pair as a whole should be evaluated with the call-by-name strategy $\mathcal{N}$. We could better reflect such a data type in Haskell with strict fields, by accounting for memoization through the call-by-need strategy, by just replacing $\mathcal{N}$ with $\mathcal{LV}$.

Remark 5.6. Recall from Section 5.2 that although the η axioms for data and co-data types do not reference the chosen strategy, their expressive power is affected by the substitution principle, which is in turn affected by the choice of values and co-values. In light of this observation, if we were forced to pick only *one* strategy for all data types and *one* strategy for all co-data types, it would make sense to pick the strategies that would give us the strongest equational theories. Therefore, if we want to make the η axiom for a data type as strong as possible, we should choose the call-by-value $\mathcal{V}$ strategy, since by substitution *every* co-term of that data type is equivalent to a case abstraction on the structure of the type. Likewise, if we want to make the η axiom for a co-data type as strong as possible, we should choose the call-by-name $\mathcal{N}$ strategy, since by substitution *every* term of that co-data type is equivalent to a co-case abstraction on the co-structure of the type. In this sense, the decision use of polarities (i.e. the data/co-data divide) to determine evaluation strategy is the same as choosing strategies to get the *strongest* and *most universal* η principles for every (co-)data type. *End remark* 5.6.

Combining Strategies in Evaluation

Now that we are looking at programs with multiple different strategies running around, we need to be able make sure that only terms and co-terms from the same strategy interact with one another. Otherwise, the same fundamental dillema that we were trying to avoid could crop back up again. For example, suppose we have a program using both the call-by-value and call-by-name strategies, $\mathcal{V}$ and $\mathcal{N}$, and face the usual problematic command $c_0 = \langle \mu\_.c_1 \| \tilde{\mu}\_.c_2 \rangle$. If we interpret the term $\mu\_.c_1$ as call-by-name then it is a value of $\mathcal{N}$, meaning it is a valid instance of $\tilde{\mu}_V$ substitution, and if we interpret the co-term $\tilde{\mu}\_.c_2$ as call-by-value, then it is a co-value of $\mathcal{V}$, meaning it is a valid instance of μ_E substitution. This puts us back where we started, where $c_1 =_{\mu_E} c_0 =_{\tilde{\mu}_V} c_2$ due to the conflict in a $\mathcal{N}$-$\mathcal{V}$ interaction. Thus, our goal is to be able to instantiate the parametric equational theory with a more complex *composite* strategy made up of several *primitive* strategies, and use the kinds of types to make sure that the terms and co-terms agree on which strategy to use in a command. This way, we can understand how to write and run programs that interleave several different evaluation strategies, and be sure that we will still get out the expected result in the end.

Recall from Chapter IV that as a way out of the dilemma, Danos *et al.* (1997) shows that we can use types to disambiguate the expected evaluation order in unclear commands. This procedure follows the assumption that η laws are *universal* (Graham-Lengrand, 2015): the η law of every (co-)data type applies to arbitrary (co-)terms of the type without restriction. However, that procedure is not directly applicable in the more general setting where the η laws are restricted to (co-)values, since we no longer assume that data types *must* follow a call-by-value order and co-data types *must* follow a call-by-name order. However, we still assume that each type, be it data or co-data, must belong to a kind specifying *some* evaluation order. Thus, we can still use a type-based approach for evaluation, albeit a more general one, by just checking the kind of the principle type of interaction in a command. In this sense, the typed $\mu\tilde{\mu}$ and $\beta\eta$ laws can already be generalized to multiple strategies $\mathcal{S}$, giving the typed $\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\to}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}$ equational theory for multi-strategy (co-)data types $\mathcal{G}$. Of note, we only need to perform the type-based strategy-lookup during μ or $\tilde{\mu}$ substitution:

$$\begin{array}{lll} (\mu_{\vec{\mathcal{S}}}) & \langle \mu\alpha.c \| E \rangle \succ_{\mu_{\vec{\mathcal{S}}}} c\{E/\alpha\} & (E : A, A : \mathcal{S}, E \in \mathit{CoValues}_{\mathcal{S}}) \\ (\tilde{\mu}_{\vec{\mathcal{S}}}) & \langle V \| \tilde{\mu}x.c \rangle \succ_{\tilde{\mu}_{\vec{\mathcal{S}}}} c\{V/x\} & (V : A, A : \mathcal{S}, V \in \mathit{Values}_{\mathcal{S}}) \end{array}$$

and otherwise restrict the rewriting rules as usual so that both sides have the same type. The type-restricted rules rely on the type associated with (co-)terms in a command to decide on the appropriate strategy for deciding values and co-values, thus fixing an priority between the opposing μ and $\tilde{\mu}$ substitution rules. In other words, we can always use typing information to evaluate a multi-strategy program without falling back into the fundamental dilemma of classical computation.

As an example, consider an application of the typed β law for the data connective $\mathsf{MixedPair}$ as defined previously in Section 5.4. Recall that the typed β laws do not make reference to the chosen strategy in any way, they are only responsible for breaking apart structures. This means that the β rules are completely unaffected by the use of composite strategies. For instance, we may simplify a program using $\mathsf{MixedPair}$ in the same way as the call-by-value $\otimes$:

$$\begin{aligned}\langle \mathsf{MPair}(v, v') \| \tilde{\mu}[\mathsf{MPair}(x : A, y : B).c] \rangle &\to_{\beta^{\mathsf{MixedPair}}} \langle v \| \tilde{\mu}x : A. \langle v' \| \tilde{\mu}y : B.c \rangle \rangle \\ &\to_{\tilde{\mu}_{\mathcal{N}}} \langle v \| \tilde{\mu}x : A.c \{v'/y : B\} \rangle\end{aligned}$$

Notice that as before, the input abstractions take over for determining evaluation order in even with multiple primitive strategies, only now the type of the command comes more directly into play. In this case, we are allowed to substitute v' for $y : B$ since $v' : B$ and $B : \mathcal{N}$, which can be found in the implied typing derivation of the command, and $\mathit{Value}_{\mathcal{N}}$ includes every term. However, we must first evaluate v before substituting it for $x : A$. The implied typing derivation tells us that $v : A$ and $A : \mathcal{V}$, so v can only be substituted by the $\tilde{\mu}_{\mathcal{V}}$ rule if it has the restrictive form of value given by $\mathit{Value}_{\mathcal{V}}$. But the input abstraction for $x : A$ likewise has the type $A : \mathcal{V}$, so it is already a co-value of $\mathit{CoTerm}_{\mathcal{V}}$.

However, since we are only interested in determinism, a full typing discipline is overkill for the untyped $\beta\varsigma$ theory of (co-)data. After all, neither the parametric core $\mu_{\mathcal{S}}\tilde{\mu}_{\int}$ theory nor the $\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}$ theory needed to use types to maintain determinism when instantiated with a single strategy $\mathcal{S}$. Therefore, we use a type-agnostic kind system for making sure that all commands are well-kinded. By "type-agnostic," we mean that we are checking the property $v :: \mathcal{S}$, that is v is a term of some unknown type of kind $\mathcal{S}$. The kind system for the structural core $\mu\tilde{\mu}$-calculus is shown in Figure 5.17, and unremarkably resembles the ordinary type system except at "one level up." The whole point of the system is shown in the *Cut* rule that only allows commands between term and co-term of the same kind, whereas (co-)variables have the kind assumed in the environment, and input and output abstractions are generic over the kind of variable they abstract. Furthermore, the additional kinding rules for generic declared (co-)data types is shown in Figure 5.18. The main property that distinguishes this from an ordinary type system is that we "forget" the types, effectively collapsing them down into a single universal type for each kind, similar to a generalized version of Zeilberger's

$$\Gamma \in \mathit{InputEnv} ::= x_1 :: \mathcal{S}_1, \dots, x_n :: \mathcal{S}_n \qquad \Delta \in \mathit{OutputEnv} ::= \alpha_1 : \mathcal{S}_1, \dots, \alpha_n : \mathcal{S}_n$$

$$\mathit{Judgement} ::= c :: \left(\Gamma \vdash_{\mathcal{G}} \Delta\right) \mid (\Gamma \vdash_{\mathcal{G}} v :: \mathcal{S} \mid \Delta) \mid (\Gamma \mid e :: \mathcal{S} \vdash_{\mathcal{G}} \Delta)$$

Core rules:

$$\frac{}{x :: \mathcal{S} \vdash_{\mathcal{G}} x :: \mathcal{S} \mid}\ \mathit{VR} \qquad \frac{}{\mid \alpha :: A \vdash_{\mathcal{G}} \alpha :: \mathcal{S}}\ \mathit{VL}$$

$$\frac{c :: \left(\Gamma \vdash_{\mathcal{G}} \alpha :: \mathcal{S}, \Delta\right)}{\Gamma \vdash_{\mathcal{G}} \mu\alpha.c :: \mathcal{S} \mid \Delta}\ \mathit{AR} \qquad \frac{c :: \left(\Gamma, x :: \mathcal{S} \vdash_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}x.c :: \mathcal{S} \vdash_{\mathcal{G}} \Delta}\ \mathit{AL}$$

$$\frac{\Gamma \vdash v :: \mathcal{S} \mid \Delta \quad \Gamma' \mid e :: \mathcal{S} \vdash \Delta'}{\langle v \| e \rangle :: (\Gamma', \Gamma \vdash \Delta', \Delta)}\ \mathit{Cut}$$

FIGURE 5.17. Type-agnostic kind system for the core $\mu\tilde{\mu}$ sequent calculus.

(2009) “bi-typed” system, where we now allow for as many base kinds as desired. This kind system is a relaxation of the full typing regime, in that all well-typed commands and (co-)terms are well-kinded by demoting the environments $x_1 : A_1, \dots, x_n : A_n$ and $\alpha_1 : B_1, \dots, \alpha_m : B_m$ to $x_1 :: \mathcal{T}_1, \dots, x_n :: \mathcal{T}_n$ and $\alpha_1 :: \mathcal{R}_1, \dots, \alpha_m : \mathcal{R}_m$, where $A_1 : \mathcal{T}_1, \dots, A_n : \mathcal{T}_n$ and $B_1 : \mathcal{R}_1, \dots, B_m : \mathcal{R}_m$ in the given typing environment.

Now that we have refined the untyped syntax into the well-kinded sub-syntax, we can build composite strategies that combine multiple primitive ones. Essentially, a composite substitution strategy is one whose values and co-values are further subdivided into different base kinds. This way, each value in a composite strategy belongs to exactly one kind of term, corresponding to the particular “primitive” strategy that it comes from. Furthermore, to get a full composite evaluation strategy, we also need to compose the evaluation contexts that come from each “primitive” strategy, to get a single set of evaluation contexts that intermingles them all. In general, we can form the composite strategy $\overrightarrow{\mathcal{S}_i}^{i} = \mathcal{S}_1, \dots, \mathcal{S}_n$ as shown in Figure 5.19. As discussed previously in Remark 5.5, each of the substitution strategies we have considered so far follows a predictable pattern, so we will first just focus on the core of the strategy without (co-)data.

For example, combining call-by-value and call-by-name into a single composite strategy is the most straightforward, and is essentially just a disjoint union of the $\mathcal{V}$ and $\mathcal{N}$ strategies, as shown in Figure 5.20, where the “disjointness” is enforced by the kinding restriction on (co-)values. Note that this combination exactly captures the

$$\text{Given } \mathbf{data}\, \mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\, \mathbf{where}\, \overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}, \text{ we have:}$$

$$\frac{\overrightarrow{\Gamma'_j \mid e :: \mathcal{R}_{ij} \vdash_{\mathcal{G}} \Delta'_j}^{j} \quad \overrightarrow{\Gamma_j \mid v :: \mathcal{T}_{ij} \vdash_{\mathcal{G}} \Delta_j}^{j}}{\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \vdash_{\mathcal{G}} \mathsf{K}_i(\overrightarrow{e}, \overrightarrow{v}) :: \mathcal{S} \mid \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}}\ \mathsf{F}R_{\mathsf{K}_i} \qquad \frac{\overrightarrow{c_i :: \left(\Gamma, \overrightarrow{x_i :: \mathcal{T}_{ij}} \vdash_{\mathcal{G}} \overrightarrow{\alpha_i :: \mathcal{R}_{ij}}, \Delta\right)}^{i}}{\Gamma \mid \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_i}, \overrightarrow{x_i}).c_i}^{i}\right] :: \mathcal{S} \vdash_{\mathcal{G}} \Delta}\ \mathsf{F}L$$

$$\text{Given } \mathbf{codata}\, \mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\, \mathbf{where}\, \overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\overrightarrow{X}) \vdash \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}, \text{ we have:}$$

$$\frac{\overrightarrow{c_i :: \left(\Gamma, \overrightarrow{x_i :: \mathcal{T}_{ij}} \vdash_{\mathcal{G}} \overrightarrow{\alpha_i :: \mathcal{R}_{ij}}, \Delta\right)}^{i}}{\Gamma \vdash_{\mathcal{G}} \mu\left(\overrightarrow{\mathsf{O}_i[\overrightarrow{x_i}, \overrightarrow{\alpha_i}].c_i}^{i}\right) :: \mathcal{S} \mid \Delta}\ \mathsf{G}R \qquad \frac{\overrightarrow{\Gamma_j \mid v :: \mathcal{T}_{ij} \vdash_{\mathcal{G}} \Delta_j}^{j} \quad \overrightarrow{\Gamma'_j \mid e :: \mathcal{R}_{ij} \vdash_{\mathcal{G}} \Delta'_j}^{j}}{\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \mid \mathsf{O}_i[\overrightarrow{v}, \overrightarrow{e}] :: \mathcal{S} \vdash_{\mathcal{G}} \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}}\ \mathsf{G}L_{\mathsf{O}_i}$$

FIGURE 5.18. Type-agnostic kind system for multi-kinded (co-)data.

$$\begin{array}{rlrl}
V \in \mathit{Value}_{\overrightarrow{\mathcal{S}_i}^{i}} ::= V_{\mathcal{S}_i} :: \mathcal{S}_i & \qquad & V_{\mathcal{S}_i} \in \mathit{Value}_{\mathcal{S}_i} ::= \dots \\
E \in \mathit{CoValue}_{\overrightarrow{\mathcal{S}_i}^{i}} ::= E_{\mathcal{S}_i} :: \mathcal{S}_i & \qquad & E_{\mathcal{S}_i} \in \mathit{CoValue}_{\mathcal{S}_i} ::= \dots \\
D \in \mathit{EvalCxt}_{\overrightarrow{\mathcal{S}_i}^{i}} ::= \square \mid D_i[D] & \qquad & D_i \in \mathit{EvalCxt}_{\mathcal{S}_i} ::= \dots
\end{array}$$

FIGURE 5.19. Composite $\overrightarrow{\mathcal{S}}$ strategy.

$$\begin{array}{ll}
V \in \mathit{Value}_{\mathcal{P}} ::= V_{\mathcal{V}} :: \mathcal{V} \mid V_{\mathcal{N}} :: \mathcal{N} & \quad E \in \mathit{CoValue}_{\mathcal{P}} ::= E_{\mathcal{V}} :: \mathcal{V} \mid E_{\mathcal{N}} :: \mathcal{N} \\
V_{\mathcal{V}} \in \mathit{Value}_{\mathcal{V}} ::= x & \quad E_{\mathcal{V}} \in \mathit{CoValue}_{\mathcal{V}} ::= e \\
V_{\mathcal{N}} \in \mathit{Value}_{\mathcal{N}} ::= v & \quad E_{\mathcal{N}} \in \mathit{CoValue}_{\mathcal{N}} ::= \alpha \\
\multicolumn{2}{l}{D \in \mathit{EvalCxt}_{\mathcal{P}} ::= \square \mid \langle \square \| e :: \mathcal{V} \rangle \mid \langle V :: \mathcal{V} \| \square \rangle \mid \langle v :: \mathcal{N} \| \square \rangle \mid \langle \square \| E :: \mathcal{N} \rangle}
\end{array}$$

FIGURE 5.20. Composite core polarized strategy $\mathcal{P} = \mathcal{V}, \mathcal{N}$.

$$
\begin{aligned}
V \in \mathit{Value}_{\mathcal{LV},\mathcal{LN}} &::= V_{\mathcal{LV}} :: \mathcal{LV} \mid V_{\mathcal{LN}} :: \mathcal{LN} \\
V_{\mathcal{LV}} \in \mathit{Value}_{\mathcal{LV}} &::= x \\
V_{\mathcal{LN}} \in \mathit{Value}_{\mathcal{LN}} &::= x \mid \mu\alpha.D[\langle V_{\mathcal{LN}} \| \alpha \rangle]
\end{aligned}
$$

$$
\begin{aligned}
E \in \mathit{CoValue}_{\mathcal{LV},\mathcal{LN}} &::= E_{\mathcal{LV}} :: \mathcal{LV} \mid E_{\mathcal{LN}} :: \mathcal{LV} \\
E_{\mathcal{LV}} \in \mathit{CoValue}_{\mathcal{LV}} &::= \alpha \mid \tilde{\mu}x.D[\langle x \| E_{\mathcal{LV}} \rangle] \\
E_{\mathcal{LN}} \in \mathit{CoValue}_{\mathcal{LN}} &::= \alpha
\end{aligned}
$$

$$
\begin{aligned}
D \in \mathit{EvalCxt} &::= \Box \mid \langle v :: \mathcal{LV} \| \tilde{\mu}x.D \rangle \mid \langle \mu\alpha.D \| e :: \mathcal{LN} \rangle \\
&\quad \mid \langle v :: \mathcal{LV} \| \Box \rangle \mid \langle \Box \| E :: \mathcal{LV} \rangle \mid \langle \Box \| e :: \mathcal{LN} \rangle \mid \langle V :: \mathcal{LN} \| \Box \rangle
\end{aligned}
$$

FIGURE 5.21. Composite core $\mathcal{LV}$ and $\mathcal{LN}$ strategy.

polarized evaluation strategy $\mathcal{P}$ for system L in Section 4.3. Combining call-by-need with its dual is a little more involved, since both the $\mathcal{LV}$ and $\mathcal{LN}$ substitution strategies form "closures" over evaluation contexts that can include delayed (co-)terms that have not yet been evaluated, but whose results should be shared. Thus, to combine these two strategies, we rely on the merged evaluation contexts of the composite strategy, as shown in Figure 5.21. Additionally, all four primitive strategies can be combined into a single composite strategy by taking the disjoint union of the previous two combinations in the expanded composite syntax, so that the $\mathcal{V}, \mathcal{N}, \mathcal{LV}, \mathcal{LN}$ strategy is defined as the following sets of (co-)values:

$$
\begin{aligned}
\mathit{Value}_{\mathcal{V},\mathcal{N},\mathcal{LV},\mathcal{LN}} &\triangleq \mathit{Value}_{\mathcal{P}} \cup \mathit{Value}_{\mathcal{LV},\mathcal{LN}} \\
\mathit{CoValue}_{\mathcal{V},\mathcal{N},\mathcal{LV},\mathcal{LN}} &\triangleq \mathit{CoValue}_{\mathcal{P}} \cup \mathit{CoValue}_{\mathcal{LV},\mathcal{LN}} \\
\mathit{EvalCxt}_{\mathcal{V},\mathcal{N},\mathcal{LV},\mathcal{LN}} &\triangleq \mathit{EvalCxt}_{\mathcal{P}} \cup \mathit{EvalCxt}_{\mathcal{LV},\mathcal{LN}}
\end{aligned}
$$

Finally, we add (co-)data to all of these composite strategies using the method described in Remark 5.5: we extend every $Value_{\mathcal{S}}$ with all well-kinded co-case abstractions and terms of the form $\mathsf{K}(\vec{E}, \vec{V})$ in $Term_{\mathcal{S}}$, extend every $CoValue_{\mathcal{S}}$ with all well-kinded case abstractions and co-terms of the form $\mathsf{O}[\vec{V}, \vec{E}]$ in $CoTerm_{\mathcal{S}}$, and close every $Value_{\mathcal{S}}$ and $CoValue_{\mathcal{S}}$ under ς expansion.

The main goal in tracking the strategy in the kinds is to continue to avoid the fundamental dilemma of classical computation when mixing strategies. Well-kindedness ensures that we cannot have a command between a term and a co-term following different primitive strategies, so that the kind restriction is enough to determine a consistent strategy for every substitution and avoid the fundamental dillema. For example, it is enough for composite strategies like $\mathcal{P}$ or $\mathcal{LV}, \mathcal{LN}$, since it lets us determine the appropriate strategy to use for every substitution, which prevents re-introducing the critical pair between μ and $\tilde{\mu}$. Furthermore, well-kindedness is preserved by the untyped reduction theory, so that we only need to begin with a well-kinded command or (co-)term to ensure that every step stays well-kinded.

Theorem 5.4 (Kind preservation)**.** *For all strategies $\vec{\mathcal{S}} = \mathcal{S}_1, \dots, \mathcal{S}_n$:*

a) If $c :: \left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$ and $c \to_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}} c'$ then $c' : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$.

b) If $\Gamma \vdash_{\mathcal{G}} v :: \mathcal{S}_i \mid \Delta$ and $v \to_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}} v'$ then $\Gamma \vdash_{\mathcal{G}} v' :: \mathcal{S}_i \mid \Delta$.

c) If $\Gamma \mid e :: \mathcal{S}_i \vdash_{\mathcal{G}} \Delta$ and $e \to_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}} e'$ then $\Gamma \mid e' :: \mathcal{S}_i \vdash_{\mathcal{G}} \Delta$.

Proof. By (mutual) induction on the kinding derivations $c :: \left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$, $\Gamma \vdash_{\mathcal{G}} v :: \mathcal{S}_i \mid \Delta$, and $\Gamma \mid e :: \mathcal{S}_i \vdash_{\mathcal{G}} \Delta$. The cases of the compatible closure of the base $\succ$ rewriting rules follow directly from the inductive hypothesis, and the base cases for the $\succ$ rewriting rules follows from the fact that well-kindedness is preserved under substitution, i.e. that for any $\Gamma' \vdash_{\mathcal{G}} V :: \mathcal{S}_i \mid \Delta'$ and $\Gamma' \mid E :: \mathcal{S}_i \vdash_{\mathcal{G}} \Delta'$,

1. $c :: \left(\Gamma, x :: \mathcal{S}_i \vdash_{\mathcal{G}} \Delta\right)$ implies $c\{V/x\} :: \left(\Gamma, \Gamma' \vdash_{\mathcal{G}} \Delta, \Delta'\right)$ and $c :: \left(\Gamma \vdash_{\mathcal{G}} \alpha :: \mathcal{S}_i, \Delta\right)$ implies $c\{E/\alpha\} :: \left(\Gamma, \Gamma' \vdash_{\mathcal{G}} \Delta, \Delta'\right)$,

2. $\Gamma, x :: \mathcal{S}_i \vdash_{\mathcal{G}} v :: \mathcal{S}_j \mid \Delta$ implies $\Gamma, \Gamma' \vdash_{\mathcal{G}} v\{V/x\} :: \mathcal{S}_j \mid \Delta, \Delta'$ and $\Gamma \vdash_{\mathcal{G}} v :: \mathcal{S}_j \mid \alpha :: \mathcal{S}_i, \Delta$ implies $\Gamma, \Gamma' \vdash_{\mathcal{G}} v\{E/\alpha\} :: \mathcal{S}_j \mid \Delta, \Delta'$, and

3. $\Gamma, x :: \mathcal{S}_i \mid e :: \mathcal{S}_j \vdash_{\mathcal{G}} \Delta$ implies $\Gamma, \Gamma' \mid e\{V/x\} :: \mathcal{S}_j \vdash_{\mathcal{G}} \Delta, \Delta'$ and $\Gamma \mid e :: \mathcal{S}_j \vdash_{\mathcal{G}} \alpha :: \mathcal{S}_i, \Delta$ implies $\Gamma, \Gamma' \mid e\{E/\alpha\} :: \mathcal{S}_j \vdash_{\mathcal{G}} \Delta, \Delta'$, and

each of which follows by induction on the kinding derivations for c, v, and e. □

This means that we can safely compute the result of any untyped command or (co-)term so long as it is well-kinded to begin with. Returning to the $\mathsf{MixedPair}$ example, if we begin with the well-kinded command $\langle \mathsf{MPair}(V, V') \| \tilde{\mu}[\mathsf{MPair}(x, y).c] \rangle$, then we know that $V :: \mathcal{V}$ and $V' :: \mathcal{N}$, so V cannot be an output abstraction but V' can be due to the kinded definition of $\mathit{Value}_{\mathcal{P}}$. This gives us the reduction

$$\langle \mathsf{MPair}(V, V') \| \tilde{\mu}[\mathsf{MPair}(x, y).c] \rangle \to_{\beta_{\mathcal{P}}} c\,\{V/x, V'/y\}$$

which induces the combined substitution of the $\mathcal{V}$-value V and $\mathcal{N}$-value V', resulting in the command $c\,\{V/x, V'/y\}$ of the same kind that we started with.

Duality of Connectives and Evaluation

Having laid out a general system for both data and co-data and with the possibility of intermingling multiple evaluation strategies, we now rephrase the duality of the sequent calculus. In particular, given any instance of the parametric $\mu\tilde{\mu}$-calculus, we are able to automatically generate its dual instance, such that the two are isomorphic to one another by the involutive duality operation. Additionally, particular application of the duality-generating operation recapitulates the previous results of duality in the sequent calculus, giving a single setting for summarizing the study of computational duality. Effectively, duality applies in both the static world of types as well as the dynamic world of programs. In types, duality expresses the opposing purpose of assumption and conclusion on the two sides of a sequent. In programs, duality expresses the opposing purpose of production and consumption on the two sides of a command. Thus, the entailment ($\vdash$) of a sequent and the dividing line of a command provide the fundamental pole about which opposing entities turn in their dance of duality.

The main difference from before is that we now have many sources for names that must be dualized. Types and programs in the parametric sequent calculus contain a variety of names—free variables and co-variables, constructors and observers, and connectives for data and co-data types. These names are arbitrary identifiers which ultimately do not impact the meaning of types or programs. However, to examine duality we must relate pairs of these arbitrary names. Therefore, we build our duality on a given relationship between dual names, written as an overline. Recall that in both the dual calculi (Chapter III) and system L (Chapter IV), duality swaps variables with co-variables, and vice versa. Formally, this is represented by an assumed bijection, $\overline{x}$ and $\overline{\alpha}$, between the two dual variable sets. But in the parametric $\mu\tilde{\mu}$-calculus, (co-)variables aren't the only names we must think about; we also have to do something about the names of connectives (F) as well as the names of constructors and observers (K and

O). Therefore, we also assume a bijection between constructors and observers, $\overline{\mathsf{K}}$ and $\overline{\mathsf{O}}$, as well as a bijection between connective names, $\overline{\mathsf{F}}$. Additionally, for multi-kinded programs we need a bijection $\overline{\mathcal{S}}$ between the names for base kinds. As shorthand, we may use the dual identifier relation $\eqsim$ which identifies the chosen duals to the various bijections, so that $x \eqsim \alpha$ means $\overline{x} = \alpha$ and $\overline{\alpha} = x$ and so on for the other namespaces.

With the bijections between names at hand, we first consider the duality of types as shown in Figure 5.22. As before, the static aspect of this duality is exactly the usual form of logical duality of the sequent calculus, where the input environment, Γ, is swapped with the output environment, Δ, in a sequent. Duality of the environments is defined pointwise, so for every variable $x : A$ we associate a dual co-variable denoted $\overline{x} : A^{\perp}$, and likewise every co-variable $\alpha : A$ is associated with a dual variable denoted $\overline{\alpha} : A^{\perp}$. Duality of the kinding environments from Figure 5.15 for multi-kinded programs is similar, except that instead of types we have base kinds, and the dual of $\mathcal{S}$ is $\overline{\mathcal{S}}$. In sequents, terms swap places with co-terms and vice versa. For example, the dual of a closed term, $\vdash_{\mathcal{G}} v : A \mid$ or $\vdash_{\mathcal{G}} v :: \mathcal{S} \mid$, is a closed co-term, $\mid v^{\perp} : A^{\perp} \vdash_{\mathcal{G}^{\perp}}$ or $\mid v^{\perp} :: \mathcal{S}^{\perp} \vdash_{\mathcal{G}^{\perp}}$. Going the other way, a type derivation of a closed co-term, $\mid e : A \vdash_{\mathcal{G}}$ or $\mid e :: \mathcal{S} \vdash_{\mathcal{G}}$, is dualized as a type derivation of a closed term, $\vdash_{\mathcal{G}^{\perp}} e^{\perp} : A^{\perp} \mid$ or $\vdash_{\mathcal{G}^{\perp}} e^{\perp} :: \mathcal{S}^{\perp} \mid$. Commands, which sit outside of the sequent, stay in place and instead describe the dynamic aspect of dualization inside a program.

Each data type declaration is dual to a co-data type declaration, and vice versa. On the one hand, the constructors, K_i, of data type declaration become the observers of the dual co-data type declaration, denoted $\overline{\mathsf{K}_i}$. On the other hand, the observers, O_i, of a co-data type declaration become the constructors of the dual data type declaration, denoted $\mathsf{O}_i^{\perp}$. Furthermore, the sequents describing each constructor are also reversed by the duality operation on sequents, similar to the action of duality on typing judgements. For example, Figure 5.6 shows several dual data and co-data declarations side-by-side, so that set of declarations is self-dual, under the following dualization relationship for names:

$$\begin{array}{ccc}
\oplus \eqsim \& & \iota_1 \eqsim \pi_1 & \iota_2 \eqsim \pi_2 \\
\otimes \eqsim \parr & (\_,\_) \eqsim [\_,\_] & \\
1 \eqsim \perp & () \eqsim [] & \\
0 \eqsim \top & & \\
- \eqsim \to & \_\cdot\_ \eqsim \_\cdot\_ &
\end{array}$$

Duality of environments:

$$
\begin{array}{cc}
(\overrightarrow{X : k})^{\perp} \triangleq \overrightarrow{X : k^{\perp}} & (\overrightarrow{decl})^{\perp} \triangleq \overrightarrow{decl^{\perp}} \\
(\overrightarrow{x : A})^{\perp} \triangleq \overrightarrow{\overline{x} : A^{\perp}} & (\overrightarrow{x :: \mathcal{S}})^{\perp} \triangleq \overrightarrow{\overline{x} :: \overline{\mathcal{S}}} \\
(\overrightarrow{\alpha : A})^{\perp} \triangleq \overrightarrow{\overline{\alpha} : A^{\perp}} & (\overrightarrow{\alpha :: \mathcal{S}})^{\perp} \triangleq \overrightarrow{\overline{\alpha_1} :: \overline{\mathcal{S}_1}}
\end{array}
$$

Duality of sequents:

$$
\begin{array}{cc}
\Big(c : \big(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\big)\Big)^{\perp} \triangleq c^{\perp} : \big(\Delta^{\perp} \vdash^{\Theta^{\perp}}_{\mathcal{G}^{\perp}} \Gamma^{\perp}\big) & \Big(c :: \big(\Gamma \vdash_{\mathcal{G}} \Delta\big)\Big)^{\perp} \triangleq c^{\perp} :: \big(\Delta^{\perp} \vdash_{\mathcal{G}^{\perp}} \Gamma^{\perp}\big) \\
\Big(\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta\Big)^{\perp} \triangleq \Delta^{\perp} \mid v^{\perp} : A^{\perp} \vdash^{\Theta^{\perp}}_{\mathcal{G}^{\perp}} \Gamma^{\perp} & \Big(\Gamma \vdash_{\mathcal{G}} v :: \mathcal{S} \mid \Delta\Big)^{\perp} \triangleq \Delta^{\perp} \mid v^{\perp} :: \overline{\mathcal{S}} \vdash_{\mathcal{G}^{\perp}} \Gamma^{\perp} \\
\Big(\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta\Big)^{\perp} \triangleq \Delta^{\perp} \vdash^{\Theta^{\perp}}_{\mathcal{G}^{\perp}} e^{\perp} : A^{\perp} \mid \Gamma^{\perp} & \Big(\Gamma \mid e :: \mathcal{S} \vdash_{\mathcal{G}} \Delta\Big)^{\perp} \triangleq \Delta^{\perp} \vdash^{\Theta^{\perp}}_{\mathcal{G}^{\perp}} e^{\perp} :: \overline{\mathcal{S}} \mid \Gamma^{\perp}
\end{array}
$$

Duality of declarations:

$$
\left(\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}} \\
\quad \mathsf{K}_1 : & \overrightarrow{A_1 : \mathcal{T}_1} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_1 : \mathcal{R}_1} \\
\quad \ldots & \\
\quad \mathsf{K}_n : & \overrightarrow{A_n : \mathcal{T}_n} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_n : \mathcal{R}_n}
\end{array}\right)^{\perp}
\triangleq
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{codata}\,\overline{\mathsf{F}}(\overrightarrow{X : k^{\perp}}) : \overline{\mathcal{S}}\,\mathbf{where}} \\
\quad \overline{\mathsf{K}_1} : & \overrightarrow{B_1^{\perp} : \mathcal{R}_1} \mid \overline{\mathsf{F}}(\overrightarrow{X}) \vdash \overrightarrow{A_1^{\perp} : \mathcal{T}_1} \\
\quad \ldots & \\
\quad \mathsf{K}_n^{\perp} : & \overrightarrow{B_n^{\perp} : \mathcal{R}_n} \mid \overline{\mathsf{F}}(\overrightarrow{X}) \vdash \overrightarrow{A_n^{\perp} : \mathcal{T}_n}
\end{array}
$$

$$
\left(\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{codata}\,\mathsf{G}(\overrightarrow{X : k^{\perp}}) : \mathcal{S}\,\mathbf{where}} \\
\quad \mathsf{O}_1 : & \overrightarrow{A_1 : \mathcal{T}_1} \mid \mathsf{G}(\overrightarrow{X}) \vdash \overrightarrow{B_1 : \mathcal{R}_1} \\
\quad \ldots & \\
\quad \mathsf{O}_n : & \overrightarrow{A_n : \mathcal{T}_n} \mid \mathsf{G}(\overrightarrow{X}) \vdash \overrightarrow{B_n : \mathcal{R}_n}
\end{array}\right)^{\perp}
\triangleq
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{data}\,\overline{\mathsf{G}}(\overrightarrow{X : k^{\perp}}) : \overline{\mathcal{S}}\,\mathbf{where}} \\
\quad \overline{\mathsf{O}_1} : & \overrightarrow{B_1^{\perp} : \mathcal{R}_1} \vdash \overline{\mathsf{G}}(\overrightarrow{X}) \mid \overrightarrow{A_1^{\perp} : \mathcal{T}_1} \\
\quad \ldots & \\
\quad \overline{\mathsf{O}_n} : & \overrightarrow{B_n^{\perp} : \mathcal{R}_n} \vdash \overline{\mathsf{G}}(\overrightarrow{X}) \mid \overrightarrow{A_n^{\perp} : \mathcal{T}_n}
\end{array}
$$

Duality of types and kinds:

$$
\mathcal{S}^{\perp} \triangleq \overline{\mathcal{S}} \qquad X^{\perp} \triangleq X \qquad \mathsf{F}(\overrightarrow{A})^{\perp} \triangleq \overline{\mathsf{F}}(\overrightarrow{A^{\perp}}) \qquad \mathsf{G}(\overrightarrow{A})^{\perp} \triangleq \overline{\mathsf{G}}(\overrightarrow{A^{\perp}})
$$

FIGURE 5.22. The duality of types of the parametric $\mu\tilde{\mu}$-calculus.

Duality of the core calculus:

$$\langle v \| e \rangle^{\perp} \triangleq \left\langle v^{\perp} \middle\| e^{\perp} \right\rangle$$

$$x^{\perp} \triangleq \overline{x} \qquad\qquad \alpha^{\perp} \triangleq \overline{\alpha}$$

$$(\mu\alpha.c)^{\perp} \triangleq \tilde{\mu}\overline{\alpha}.c^{\perp} \qquad\qquad [\tilde{\mu}x.c]^{\perp} \triangleq \mu\overline{x}.c^{\perp}$$

Duality of data and co-data:

$$\mathsf{K}(\vec{e}, \vec{v})^{\perp} \triangleq \overline{\mathsf{K}}[\overrightarrow{e^{\perp}}, \overrightarrow{v^{\perp}}] \qquad \tilde{\mu}[\mathsf{K}(\vec{\alpha}, \vec{x}).c \mid \ldots]^{\perp} \triangleq \mu\Big(\overline{\mathsf{K}}[\vec{\overline{\alpha}}, \vec{\overline{x}}].c^{\perp} \mid \cdots\Big)$$

$$\mathsf{O}[\vec{v}, \vec{e}]^{\perp} \triangleq \overline{\mathsf{O}}(\overrightarrow{v^{\perp}}, \overrightarrow{e^{\perp}}) \qquad \mu(\mathsf{O}[\vec{x}, \vec{\alpha}[.c \mid \ldots)^{\perp} \triangleq \tilde{\mu}\Big[\overline{\mathsf{O}}(\vec{\overline{x}}, \vec{\overline{\alpha}}).c^{\perp} \mid \cdots\Big]$$

FIGURE 5.23. The duality of programs of the parametric $\mu\tilde{\mu}$-calculus.

$\sim \overline{\sim} \neg$ $\qquad\qquad$ $\sim \overline{\sim} \neg$

The duality between types is defined inductively on the structure of the types, such that all data connectives F are replaced with their dual co-data connectives $\overline{\mathsf{F}}$, as described above, and vice versa.

Next, we move on to consider the effect of duality on programs as shown in Figure 5.23. In the core of the $\mu\tilde{\mu}$-calculus, every command $\langle v \| e \rangle$ is dual to another command representing the flipped version of itself $\left\langle v^{\perp} \middle\| e^{\perp} \right\rangle$, variables are dual to co-variables, and input abstractions and output abstractions are dual to one another. On the constructive side of data and co-data, every data structure $\mathsf{K}(\vec{e}, \vec{v})$ dual is a co-data observation $\overline{\mathsf{K}}[\overrightarrow{e^{\perp}}, \overrightarrow{v^{\perp}}]$ and every co-data observation $\mathsf{O}[\vec{v}, \vec{e}]$ is dual to a data structure $\overline{\mathsf{O}}(\overrightarrow{v^{\perp}}, \overrightarrow{e^{\perp}})$. On the destructive side of data and co-data, every case analysis on a data structure is dual to a co-data object, and every co-data object is dual to a case analysis on a data structure.

Example 5.1. Let's consider how to swap the results of a product:

$$swap_x \triangleq \mu(\pi_1[\alpha].\langle x \| \pi_2[\alpha] \rangle \mid \pi_2[\beta].\langle x \| \pi_1[\beta] \rangle)$$

$$swap_{x,\gamma} \triangleq \langle swap_x \| \gamma \rangle \triangleq \langle \mu(\pi_1[\alpha].\langle x \| \pi_2[\alpha] \rangle \mid \pi_2[\beta].\langle x \| \pi_1[\beta] \rangle) \| \gamma \rangle$$

Given that x stands for a value of $B \mathbin{\&} A$, then $swap_x$ is a term of $A \mathbin{\&} B$ such that whenever we ask for the π_1 of $swap_x$ we get the π_2 of x, and whenever we ask for the

π_2 of swap_x, we get the π_1 of x. The command $\mathit{swap}_{x,\gamma}$ then represents a program that sends the request $\gamma : A \mathbin{\&} B$ to the swapped product swap_x.

The duality of the sequent calculus lets us turn this program around, so that we are calculating with data instead of co-data. First, we need to specify how names are treated in order to generate the dualized program. For the connectives and (co-)constructors, we use the naming convention relating products (&) and sums ($\oplus$)

$$\oplus \eqsim \& \qquad\qquad \iota_1 \eqsim \pi_1 \qquad\qquad \iota_2 \eqsim \pi_2$$

along with the following bijection between the variables and co-variables involved:

$$x \eqsim \alpha' \qquad\qquad x' \eqsim \alpha \qquad\qquad y' \eqsim \beta \qquad\qquad z' \eqsim \gamma$$

What we get out from duality is then a program that swaps an injection:

$$\begin{aligned} \mathit{swap}_x^\perp &\triangleq \tilde{\mu}[\iota_1(x').\langle \iota_2(x') \| \alpha' \rangle \mid \iota_2(y').\langle \iota_1(y') \| \alpha' \rangle] \\ \mathit{swap}_{x,\gamma}^\perp &\triangleq \left\langle z' \middle\| \mathit{swap}_x^\perp \right\rangle \triangleq \langle z' \| \tilde{\mu}[\iota_1(x').\langle \iota_2(x') \| \alpha' \rangle \mid \iota_2(y').\langle \iota_1(y') \| \alpha' \rangle] \rangle \end{aligned}$$

In particular, z' stands for a value of type $A^\perp \oplus B^\perp$, and α' stands for a co-value of type $B^\perp \oplus A^\perp$. The co-term $\mathit{swap}_x^\perp$ consumes an input of type $A^\perp \oplus B^\perp$ and swaps the injection tag, turning $\iota_1(x')$ into $\iota_2(x')$ or turning $\iota_2(y')$ into $\iota_1(y')$, in order to pass a value of type $B^\perp \oplus A^\perp$ along to α'. The whole command $\mathit{swap}_{x,\gamma}^\perp$ then feeds z' into the consumer $v_0^\perp$. Notice how even though the roles of input and output have been exchanged by the duality operation, so that requests become results, the overall structure in the dual program follows the same pattern as before. *End example* 5.1.

The final piece to the puzzle is to determine the effect of duality on the strategy parameter(s) to the parametric $\mu\tilde{\mu}$-calculus. Fortunately, this duality is straightforward, since the strategy $\mathcal{S}$ is just a set of terms and co-terms (the (co-)values of the substitution strategy component of $\mathcal{S}$) and contexts (the evaluation contexts of $\mathcal{S}$). Thus, this final duality is achieved by applying the defined duality operation pointwise. Given a substitution strategy $\mathcal{S}$ whose values are given by the set $\mathit{Value}_{\mathcal{S}}$ and co-values are given by the set $\mathit{CoValue}_{\mathcal{S}}$, then we can automatically generate the dual substitution strategy $\mathcal{S}^\perp$ by swapping values with co-values, so that the values,

$\mathit{Value}_{\mathcal{S}^\perp}$, and co-values, $\mathit{CoValue}_{\mathcal{S}^\perp}$, of $\mathcal{S}^\perp$ are defined as:

$$\mathit{Value}_{\mathcal{S}^\perp} \triangleq \{E^\perp \mid E \in \mathit{CoValue}_{\mathcal{S}}\} \qquad \mathit{CoValue}_{\mathcal{S}^\perp} \triangleq \{V^\perp \mid V \in \mathit{Value}_{\mathcal{S}}\}$$

Additionally, for a full evaluation strategy $\mathcal{S}$, we can automatically generate the dual evaluation strategy by dualizing the substitution strategy component of $\mathcal{S}$ as well as its evaluation contexts $\mathit{EvalCxt}_{\mathcal{S}}$ as follows:

$$\mathit{EvalCxt}_{\mathcal{S}^\perp} \triangleq \{D^\perp \mid D \in \mathit{EvalCxt}_{\mathcal{S}}\}$$

where the duality operation is generalized to contexts in the obvious way by taking $\square^\perp = \square$. For example, dualizing the call-by-value strategy $\mathcal{V}$ generates the call-by-name strategy $\mathcal{N}$ and vice versa, and similarly for the call-by-need strategy $\mathcal{LV}$ and its dual:

$$\mathcal{V}^\perp = \mathcal{N} \qquad \mathcal{N}^\perp = \mathcal{V} \qquad\qquad \mathcal{LV}^\perp = \mathcal{LN} \qquad \mathcal{LN}^\perp = \mathcal{LV}$$

Also, the unrestricted strategy $\mathcal{U}$ is self-dual, so that $\mathcal{U}^\perp = \mathcal{U}$.

With all the dualities in place, we can now verify that the duality operation satisfies the properties we would expect. Firstly, the duality operation is involutive at all levels, so that the double-dual is an identity operation for any chosen bijection between dual namespaces.

Theorem 5.5 (Involutive duality). *The $\_^\perp$ operation on environments, sequents, declarations, types, commands, and (co-)terms is involutive, so that $\_^{\perp\perp}$ is the identity transformation.*

Proof. By (mutual) induction on the definition of the duality operation $\_^\perp$, where each case follows immediately by the inductive hypothesis. □

Secondly, the duality operation respects the static semantics of the parametric $\mu\tilde{\mu}$-calculus, so that typing of commands and (co-)terms is preserved.

Theorem 5.6 (Static duality). *If the typing judgement J (from Figures 5.8, 5.15, 5.16, 5.17 and 5.18) is derivable then $J^\perp$ is.*

Proof. By induction on the derivation of J, where in each case we must show that for some conclusion J', premises $H_1, \dots, H_n$, and inference rule I, the derivation of

$$\frac{H_1 \quad \dots \quad H_n}{J'}\ I$$

and the inductive hypothesized derivations of $H_1^\perp, \dots, H_n^\perp$ implies the derivation of

$$\frac{H_1^\perp \quad \dots \quad H_n^\perp}{J'^\perp}\ I^\perp$$

where $I^\perp$ is the dual inference rule to I, which we define as follows for both the type and kind system for programs

$$\begin{array}{llllll}
VR^\perp \triangleq VL & VL^\perp \triangleq VR & AR^\perp \triangleq AL & AL^\perp \triangleq AR & Cut^\perp \triangleq Cut & \\
\mathsf{F}R_{\mathsf{K}}^\perp \triangleq \overline{\mathsf{F}}L_{\overline{\mathsf{K}}} & \mathsf{G}L_{\mathsf{O}}^\perp \triangleq \overline{\mathsf{G}}R_{\overline{\mathsf{O}}} & \mathsf{F}L^\perp \triangleq \overline{\mathsf{F}}R & \mathsf{G}R^\perp \triangleq \overline{\mathsf{G}}L & & \\
WR^\perp \triangleq WL & WL^\perp \triangleq WR & CR^\perp \triangleq CL & CL^\perp \triangleq CR & XR^\perp \triangleq XL & XL^\perp \triangleq XR
\end{array}$$

and the kind system for types

$$data^\perp \triangleq codata \qquad codata^\perp \triangleq data \qquad TV^\perp \triangleq TV \qquad \mathsf{F}T^\perp \triangleq \overline{\mathsf{F}}T$$

The cases for the left and right rules of (co-)data ($\mathsf{F}R$ and $\mathsf{F}L$) follow from the inductive hypotheses and the fact that substitution of types commutes with duality ($A^\perp \left\{B^\perp/X\right\} =_\alpha A\left\{B/X\right\}^\perp$), which is guaranteed because the duality operation is compositional and hygienic (Downen & Ariola, 2014a). The rest of the cases follow immediately from the inductive hypotheses. □

Thirdly, the duality operation respects the dynamic aspect of the parametric $\mu\tilde{\mu}$-calculus, so that it preserves the rewriting rules between commands and (co-)terms.

Theorem 5.7 (Equational duality)**.** *For any (possibly composite) strategy $\mathcal{S}$ and set of declarations $\mathcal{G}$,*

a) if $c \succ_{R^{\mathcal{G}}_{\mathcal{S}}} c'$ then $c^\perp \succ_{R^{\mathcal{G}^\perp}_{\mathcal{S}^\perp}} c'^\perp$,

b) if $v \succ_{R^{\mathcal{G}}_{\mathcal{S}}} v'$ then $v^\perp \succ_{R^{\mathcal{G}^\perp}_{\mathcal{S}^\perp}} v'^\perp$, and *c) if* $e \succ_{R^{\mathcal{G}}_{\mathcal{S}}} e'$ *then* $e^{\perp} \succ_{R^{\mathcal{G}^{\perp}}_{\mathcal{S}^{\perp}}} e'^{\perp}$,

whenever $R^{\mathcal{G}}_{\mathcal{S}} = \mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}$, $R^{\mathcal{G}}_{\mathcal{S}} = \beta^{\mathcal{G}}\eta^{\mathcal{G}}$, *or* $R^{\mathcal{G}}_{\mathcal{S}} = \beta_{\mathcal{S}}\varsigma_{\mathcal{S}}$.

Proof. By cases on each possible rewriting rules, using the more specific fact that

a) if $c \succ_{R} c'$ then $c^{\perp} \succ_{R^{\perp}} c'^{\perp}$,

b) if $v \succ_{R} v'$ then $v^{\perp} \succ_{R^{\perp}} v'^{\perp}$, and

c) if $e \succ_{R} e'$ then $e^{\perp} \succ_{R^{\perp}} e'^{\perp}$,

where the dual of each rewriting rule R is defined as follows:

$$\begin{array}{llll}
\mu_{\mathcal{S}}{}^{\perp} \triangleq \tilde{\mu}_{\mathcal{S}^{\perp}} & \tilde{\mu}_{\mathcal{S}}^{\perp} \triangleq \mu_{\mathcal{S}^{\perp}} & \eta_{\mu}{}^{\perp} \triangleq \eta_{\tilde{\mu}} & \eta_{\tilde{\mu}}{}^{\perp} \triangleq \eta_{\mu} \\
\beta^{\mathcal{G}\perp} \triangleq \beta^{\mathcal{G}^{\perp}} & \eta^{\mathcal{G}\perp} \triangleq \eta^{\mathcal{G}^{\perp}} & & \\
\beta_{\mathcal{S}}{}^{\perp} \triangleq \beta_{\mathcal{S}^{\perp}} & \varsigma_{\mathcal{S}}{}^{\perp} \triangleq \varsigma_{\mathcal{S}^{\perp}} & &
\end{array}$$

Each case follows by the definition of the rewriting rules, the definition of the duality operation on strategies $\mathcal{S}$ and declarations $\mathcal{G}$, and the fact that substitution commutes with the duality operation (that $c^{\perp}\left\{V^{\perp}/\overline{x}\right\} =_{\alpha} (c\left\{V/x\right\})^{\perp}$, $c^{\perp}\left\{E^{\perp}/\overline{\alpha}\right\} =_{\alpha} (c\left\{E/\alpha\right\})^{\perp}$, and similarly for (co-)terms) which is guaranteed by the fact that the duality operation is compositional and hygienic (Downen & Ariola, 2014a). □

Remark 5.7. Note that the duality operation discussed here does not just compare two existing languages, as in previous work on computational duality (Curien & Herbelin, 2000; Wadler, 2003), but it actively generates the dual language to any instance of the parametric sequent calculus. Thus, we can use this operation to create the dual to any strategy of our choice. For example, applying the duality operation to the call-by-need strategy $\mathcal{LV}$ from Figures 5.3 and 5.10 generates the dual to call-by-need evaluation from Figures 5.4 and 5.10. Intuitively, the dual of call-by-need delays computation of consumers and prioritizes producers. Then we switch attention to a consumer only when we have a value to return to it. However, we do not copy complex consumers, like the way control operators in Scheme-like languages copy arbitrary call-stacks. Rather, we memoize such call-stacks, so that control operations cannot duplicate extra work inside of a continuation. And in fact, this is essentially how the "lazy call-by-name" evaluation strategy was developed by Ariola *et al.* (2011). The parametric $\mu\tilde{\mu}$-calculus generalizes the procedure to any starting evaluation strategy. *End remark* 5.7.

A (De-)Construction of the Dual Calculi

We have now seen a general language of the sequent calculus for studying a wide variety of types. Each type is characterized by two actions: building up a structure by *construction*, and analyzing the shape of a structure by *deconstruction*. The types are primarily categorized by the way they orient these actions along the producer-consumer protocol: data types produce via construction and consume via deconstruction, and co-data types produce via deconstruction and consume via construction. This viewpoint aligns neatly with system L from Chapter IV. In fact, polarized system L corresponds exactly to the $\mathcal{P}$ instance of the parametric $\mu\tilde{\mu}$-calculus with the (co-)data type declarations from Figures 5.13 and 5.14. It also aligns with the treatment of functions and polymorphism in the dual calculi: implication and the universal quantifier are both co-data types (in both call-by-value and call-by-name) that have constructed call-stacks and deconstructive λ- and $\Lambda-$abstractions, whereas the existential quantifier is a data type with constructed packages and deconstructive $\tilde{\Lambda}$-abstractions. However, the rest of the types in the dual calculi do not seem to follow this pattern: both the terms and co-terms of every type appear to be constructed, with no deconstructive pattern-matching to be found.

As it turns out, however, even the dual calculi's construction-oriented sequent calculus still follows the construction-deconstruction discipline, albeit indirectly. More formally, the simply-typed sub-language of the dual calculi (i.e. without the quantifiers) and the appropriate instances of the parametric $\mu\tilde{\mu}$-calculus are in *equational correspondence* (Sabry & Felleisen, 1992) with one another. This means that every command and (co-)term of the dual calculi can be translated to the $\mu\tilde{\mu}$-calculus, and vice versa, such that the two translations are inverses of each other, up to the equational theory, and the equations of each calculus are preserved by translation. In other words, the dual calculi can be seen as syntactic sugar by macro-expansion for a particular use-case of the $\mu\tilde{\mu}$-calculus.

Since the dual calculi really stands for a pair of two separate but dual sequent calculi—one for call-by-value and one for call-by-name—we need two translations into two different instances of the parametric $\mu\tilde{\mu}$-calculus. Because we have several representations of conjunction, disjunction, and negation as (co-)data types in the $\mu\tilde{\mu}$-calculus, as shown in Figure 5.6, our task requires us to determine which particular types correspond to the dual calculi's characterization in both call-by-value and call-by-name. Furthermore, since we aim to achieve an equational correspondence, our choice of (co-)data types must respect both the computational (β rules) *and* extensional (η rules) aspects of the types found in the dual sequent calculi.

First, let's focus on the call-by-value half of the dual calculi. To represent call-by-value conjunction, we will use the $A \otimes B$ data type. On the one hand, the terms for conjunction, (v_1, v_2), translate directly to a constructed pair of $A \otimes B$. On the other hand, the co-terms for conjunction, $\pi_1[e]$ and $\pi_2[e]$, need to be expressed as the basic deconstructions on an input of type $A \otimes B$ which extract one component of a pair:

$$\pi_1[e] \approx \tilde{\mu}[(x, \_).\langle x \| e \rangle]$$
$$\pi_2[e] \approx \tilde{\mu}[(\_, y).\langle y \| e \rangle]$$

The representation of call-by-value disjunction is similar, for which we use the $A \oplus B$ data type. On the one hand, the terms for disjunction, $\iota_1(v)$ and $\iota_2(v)$, translate directly to the constructed values of the sum type $A \oplus B$. On the other hand, the co-terms for disjunction, $[e_1, e_2]$, need to be expressed as the basic deconstruction on an input of type $A \oplus B$ which checks which of the two constructors was used:

$$[e_1, e_2] \approx \tilde{\mu}[\iota_1(x).\langle x \| e_1 \rangle \mid \iota_2(y).\langle y \| e_2 \rangle]$$

Finally, we represent the call-by-value negation with the function-like co-data type $\neg A$. On the one hand, the terms for negation, $\mathsf{not}(e)$, need to be expressed as the basic deconstruction on an output of type $\neg A$:

$$\mathsf{not}(e) \approx \mu(\neg\,[x].\langle x \| e \rangle)$$

On the other hand, the co-terms for negation, $\mathsf{not}[v]$, translate directly to the constructed co-values of the type $\neg A$. Intuitively, the role of negation in the call-by-value half of the dual calculi is to represent functions from the call-by-value λ-calculus, as used by Wadler's (2003) call-by-value encoding. Thus, we choose the form of negation that most resembles functions: the values of $\neg A$ are function-like abstractions that accept an input but do not return a result.

Having seen how to embed the call-by-value half of the dual calculi into the $\mu\tilde{\mu}_{\mathcal{V}}^{\oplus,\otimes,\neg,\rightarrow}$-calculus, we also need to translate back. As before, the constructed terms of $A \otimes B$ and $A \oplus B$, as well as the constructed co-terms of $\neg A$, translate directly. The only interesting part of the translation is in encoding deconstruction as the constructive forms. Translating the deconstructive terms of $\neg A$ is straightforward, and only requires us to place a generic input abstraction inside of the negation constructor:

$$\mu(\neg\,[x].c) \approx \mathsf{not}(\tilde{\mu}x.c)$$

Likewise, translating the deconstructive co-terms of $A \oplus B$ requires us to form a co-term pair of two generic input abstractions:

$$\tilde{\mu}[\iota_1(x).c_1 \mid \iota_2(y).c_2] \approx [\tilde{\mu}x.c_1, \tilde{\mu}y.c_2]$$

Translating a deconstructive co-term of $A \otimes B$ is the most involved, since it requires us to *copy* its input in order to extract both the first and second components one at a time. This can be achieved by naming its input with an input abstraction, and using both π_1 and π_2 on it:

$$\tilde{\mu}[(x,y).c] \approx \tilde{\mu}z.\,\langle z \| \pi_1[\tilde{\mu}x.\,\langle z \| \pi_2[\tilde{\mu}y.c]\rangle]\rangle$$

The full translation between call-by-value half of the dual calculi and the $\mu\tilde{\mu}_{\mathcal{V}}^{\oplus,\otimes,\neg,\rightarrow}$ instance of the parametric sequent calculus is shown in Figure 5.24.

Second, let's consider the call-by-name half of the dual calculi. Contrary to the call-by-value case, we will choose the opposite representations of conjunction, disjunction and negation from the (co-)data types listed in Figure 5.6: conjunction is $A \mathbin{\&} B$, disjunction is $A \mathbin{⅋} B$, and negation is $\sim A$. Likewise, the translations follow an opposite story as before: the co-terms of $A \mathbin{\&} B$ and $A \mathbin{⅋} B$ and terms of $\sim A$ translate directly, whereas the terms of $A \mathbin{\&} B$ and $A \mathbin{⅋} B$ and co-terms of $\sim A$ require more work. The disjunctive and conjunctive terms for the call-by-name calculus are translated as:

$$\begin{aligned}
\iota_1(v) &\approx \mu([\alpha, \_].\langle \alpha \| v\rangle) \\
\iota_2(v) &\approx \mu([\_, \beta].\langle \beta \| v\rangle) \\
(v_1, v_2) &\approx \mu(\pi_1[\alpha].\langle v_1 \| \alpha\rangle \mid \pi_2[\beta].\langle v_2 \| \beta\rangle)
\end{aligned}$$

and the negative co-terms for the call-by-name calculus are translated as:

$$\mathsf{not}[v] \approx \tilde{\mu}[\sim(\alpha).\langle v \| \alpha\rangle]$$

$$\langle v \| e \rangle^{\star}_{v} \triangleq \langle v^{\star}_{v} \| e^{\star}_{v} \rangle$$

$$\begin{array}{rlrl}
x^{\star}_{v} & \triangleq x & \alpha^{\star}_{v} & \triangleq \alpha \\
(\mu\alpha.c)^{\star}_{v} & \triangleq \mu\alpha.c^{\star}_{v} & [\tilde{\mu}x.c]^{\star}_{v} & \triangleq \tilde{\mu}x.c^{\star}_{v} \\
\iota_i(v)^{\star}_{v} & \triangleq \iota_i(v^{\star}_{v}) & \pi_i[e]^{\star}_{v} & \triangleq \tilde{\mu}[(x_1, x_2).\langle x_i \| e^{\star}_{v} \rangle] \\
(v_1, v_2)^{\star}_{v} & \triangleq ({v_1}^{\star}_{v}, {v_2}^{\star}_{v}) & [e_1, e_2]^{\star}_{v} & \triangleq \tilde{\mu}[\iota_1(x).\langle x \| {e_1}^{\star}_{v} \rangle \mid \iota_2(y).\langle y \| {e_2}^{\star}_{v} \rangle] \\
\mathsf{not}(e)^{\star}_{v} & \triangleq \mu(\neg\,[x].\langle x \| e^{\star}_{v} \rangle) & \mathsf{not}[v]^{\star}_{v} & \triangleq \neg\,[v^{\star}_{v}] \\
(\lambda x.v)^{\star}_{v} & \triangleq \mu([x \cdot \beta].\langle v^{\star}_{v} \| \beta \rangle) & [v \cdot e]^{\star}_{v} & \triangleq v^{\star}_{v} \cdot e^{\star}_{v}
\end{array}$$

$$\langle v \| e \rangle^{v}_{\star} \triangleq \langle v^{v}_{\star} \| e^{v}_{\star} \rangle$$

$$\begin{array}{rlrl}
x^{v}_{\star} & \triangleq x & \alpha^{v}_{\star} & \triangleq \alpha \\
(\mu\alpha.c)^{v}_{\star} & \triangleq \mu\alpha.c^{v}_{\star} & [\tilde{\mu}x.c]^{v}_{\star} & \triangleq \tilde{\mu}x.c^{v}_{\star} \\
\iota_i(v)^{v}_{\star} & \triangleq \iota_i(v^{v}_{\star}) & \tilde{\mu}[\iota_1(x).c_1 \mid \iota_2(y).c_2]^{v}_{\star} & \triangleq [\tilde{\mu}x.{c_1}^{v}_{\star}, \tilde{\mu}y.{c_2}^{v}_{\star}] \\
(v_1, v_2)^{v}_{\star} & \triangleq ({v_1}^{v}_{\star}, {v_2}^{v}_{\star}) & \tilde{\mu}[(x, y).c]^{v}_{\star} & \triangleq \tilde{\mu}z.\,\langle z \| \pi_1[\tilde{\mu}x.\,\langle z \| \pi_2[\tilde{\mu}y.c^{v}_{\star}] \rangle] \rangle \\
\mu(\neg\,[x].c)^{v}_{\star} & \triangleq \mathsf{not}(\tilde{\mu}x.c^{v}_{\star}) & \neg\,[v]^{v}_{\star} & \triangleq \mathsf{not}[v^{v}_{\star}] \\
\mu([x \cdot \beta].c)^{v}_{\star} & \triangleq \lambda x.\mu\beta.c^{v}_{\star} & [v \cdot e]^{v}_{\star} & \triangleq v^{v}_{\star} \cdot e^{v}_{\star}
\end{array}$$

FIGURE 5.24. Translation between the call-by-value half of the simply-typed dual calculi and $\mu\tilde{\mu}_{\mathcal{V}}^{\oplus,\otimes,\neg,\rightarrow}$.

Going the other way, the deconstructive co-term of $\sim A$ is translated as a negated output abstraction:

$$\tilde{\mu}[\sim(\alpha).c] \approx \mathsf{not}[\mu\alpha.c]$$

and the deconstructive term of $A \mathbin{\&} B$ is translated as a pair of output abstractions:

$$\mu(\pi_1[\alpha].c_1 \mid \pi_2[\beta].c_2) \approx (\mu\alpha.c_1, \mu\beta.c_2)$$

As before with call-by-value conjunction, translating terms of call-by-name disjunction is more involved in the dual way, requiring us to *copy* the output in order to extract both components one at a time. This can be achieved by naming its output with an output abstraction and using both ι_1 and ι_2 on it:

$$\mu([\alpha, \beta].c) \approx \mu\gamma.\, \langle \iota_1(\mu\alpha.\, \langle \iota_2(\mu\beta.c) \| \gamma \rangle) \| \gamma \rangle$$

The full translation between the call-by-name half of the dual calculi and the $\mu\tilde{\mu}_{\mathcal{N}}^{\&,⅋,\sim,\to}$ instance of the parametric sequent calculus is shown in Figure 5.25.

With the full translations to and from the dual calculi and instances of the parametric $\mu\tilde{\mu}$-calculus, we have a correspondence between the dual calculi respecting their equational theories. In particular, the $\beta\eta$ theory of (co-)data in the parametric $\mu\tilde{\mu}$-calculus corresponds to an appropriate $\beta\eta\varsigma$ theory for the dual calculi. To that point, we need to extend the dual calculi with η laws as well as with additional values in call-by-value and additional co-values in call-by-name as shown in Figure 5.26, which is based on the semantics for the dual calculi by Wadler (2005). The extra values in $\mathcal{V}'$ extend those in $\mathcal{V}$ to say that the result of projecting out of a value is itself a value, which makes intuitive sense by the meaning of call-by-value. The extra co-values in $\mathcal{N}'$ extend those in $\mathcal{N}$ to say that forcing a tagged injection also forces its payload, which may not be so obvious intuitively, but is still semantically sound by the interpretation of sum types in the call-by-name dual calculus. With this extension to the dual calculi, we get an equational correspondence.

Theorem 5.8. – *The call-by-value half of the simply-typed dual calculi is in equational correspondence with the $\mu\tilde{\mu}_{\mathcal{V}}^{\oplus,\otimes,\neg,\to}$-calculus.*

– *The call-by-name half of the simply-typed dual calculi is in equational correspondence with the $\mu\tilde{\mu}_{\mathcal{N}}^{\&,⅋,\sim,\to}$-calculus.*

$$\langle v \| e \rangle_n^\star \triangleq \langle v_n^\star \| e_n^\star \rangle$$

$$\begin{array}{rclrcl}
x_n^\star & \triangleq & x & \alpha_n^\star & \triangleq & \alpha \\
(\mu\alpha.c)_n^\star & \triangleq & \mu\alpha.c_n^\star & [\tilde{\mu}x.c]_n^\star & \triangleq & \tilde{\mu}x.c_n^\star \\
\iota_i(v)_n^\star & \triangleq & \mu([\alpha_1, \alpha_2].\langle v \| \alpha_i \rangle) & \pi_i[e]_n^\star & \triangleq & \pi_i[e_n^\star] \\
(v_1, v_2)_n^\star & \triangleq & \mu(\pi_1[\alpha].\langle v_1 \| \alpha \rangle \mid \pi_2[\beta].\langle v_2 \| \beta \rangle) & [e_1, e_2]_n^\star & \triangleq & [{e_1}_n^\star, {e_2}_n^\star] \\
\mathsf{not}(e)_n^\star & \triangleq & \sim(e_n^\star) & \mathsf{not}[v]_n^\star & \triangleq & \tilde{\mu}[\sim(\alpha).\langle v \| \alpha \rangle] \\
(\lambda x.v)_n^\star & \triangleq & \mu([x \cdot \beta].\langle v_v^\star \| \beta \rangle) & [v \cdot e]_n^\star & \triangleq & v_n^\star \cdot e_n^\star
\end{array}$$

$$\langle v \| e \rangle_\star^n \triangleq \langle v_\star^n \| e_\star^n \rangle$$

$$\begin{array}{rclrcl}
x_\star^n & \triangleq & x & \alpha_\star^n & \triangleq & \alpha \\
(\mu\alpha.c)_\star^n & \triangleq & \mu\alpha.c_\star^n & [\tilde{\mu}x.c]_\star^n & \triangleq & \tilde{\mu}x.c_\star^n \\
\mu(\pi_1[\alpha].c_1 \mid \pi_2[\beta].c_2)_\star^n & \triangleq & (\mu\alpha.{c_1}_\star^n, \mu\beta.{c_2}_\star^n) & \pi_i[e]_\star^n & \triangleq & \pi_i[e_\star^n] \\
\mu([\alpha, \beta].c)_\star^n & \triangleq & \mu\gamma.\, \langle \iota_1(\mu\alpha.\, \langle \iota_2(\mu\beta.c_\star^n) \| \gamma \rangle) \| \gamma \rangle & [e_1, e_2]_\star^n & \triangleq & [{e_1}_\star^n, {e_2}_\star^n] \\
\sim(e)_\star^n & \triangleq & \mathsf{not}(e_\star^n) & \tilde{\mu}[\sim(\alpha).c]_\star^n & \triangleq & \mathsf{not}[\tilde{\mu}x.c_\star^n] \\
\mu([x \cdot \beta].c)_\star^n & \triangleq & \lambda x.\mu\beta.c_\star^n & [v \cdot e]_\star^n & \triangleq & v_\star^n \cdot e_\star^n
\end{array}$$

FIGURE 5.25. Translation between the call-by-name half of the simply-typed dual calculi and $\mu\tilde{\mu}_{\mathcal{N}}^{\&,⅋,\sim,\rightarrow}$.

Call-by-value extended values ($\mathcal{V}'$):

$$V \in \mathit{Value}_{\mathcal{V}'} ::= \ldots \mid \mu\alpha.\, \langle V \| \pi_1\, [\alpha] \rangle \mid \mu\alpha.\, \langle V \| \pi_2\, [\alpha] \rangle$$

Call-by-name extended co-values ($\mathcal{N}'$):

$$E \in \mathit{CoValue}_{\mathcal{N}'} ::= \ldots \mid \tilde{\mu}x.\, \langle \iota_1\, (x) \| E \rangle \mid \tilde{\mu}x.\, \langle \iota_2\, (x) \| E \rangle$$

η laws for both call-by-value ($\mathcal{S} = \mathcal{V}'$) and call-by-name ($\mathcal{S} = \mathcal{N}'$):

$$\begin{array}{lcr}
(\eta_{\mathcal{S}}^{\times}) & V : A \times B \prec_{\eta_{\mathcal{S}}^{\times}} (\mu\alpha.\, \langle V \| \pi_1\, [\alpha] \rangle, \mu\beta.\, \langle V \| \pi_2\, [\beta] \rangle) & (\alpha, \beta \notin FV(V)) \\
(\eta_{\mathcal{S}}^{\oplus}) & E : A \oplus B \prec_{\eta_{\mathcal{S}}^{\oplus}} [\tilde{\mu}x.\, \langle \iota_1\, (x) \| E \rangle, \tilde{\mu}y.\, \langle \iota_2\, (y) \| E \rangle] & (x, y \notin FV(E)) \\
(\eta_{\mathcal{S}}^{\neg}) & V : \neg A \prec_{\eta_{\mathcal{S}}^{\neg}} \mathsf{not}(\tilde{\mu}x.\, \langle V \| \mathsf{not}[x] \rangle) & (x \notin FV(V)) \\
(\eta_{\mathcal{S}}^{\rightarrow}) & V : A \rightarrow B \prec_{\eta_{\mathcal{S}}^{\rightarrow}} \lambda x.\mu\beta.\, \langle V \| x \cdot \beta \rangle & (x, \beta \notin FV(V))
\end{array}$$

FIGURE 5.26. The η laws for the dual calculi and extended (co-)values ($\mathcal{V}', \mathcal{N}'$).

Proof. a) To demonstrate the call-by-value equational correspondence, we must prove the following conditions

(1) The translations $(\_)^{\star}_{v}$ and $(\_)^{v}_{\star}$ are *inverses* up to the respective equational theories of the two calculi: $c^{v\star}_{\star v} = c$ in the $\mu\tilde{\mu}^{\oplus,\otimes,\neg,\to}_{\mathcal{V}}$-calculus and $c^{\star v}_{v\star} = c$ in the call-by-value dual calculus, and similarly for (co-)terms.

(2) The two equational theories are *sound* under translation with respect to each other: $c = c'$ in the $\mu\tilde{\mu}^{\oplus,\otimes,\neg,\to}_{\mathcal{V}}$-calculus implies $c^{v}_{\star} = c'^{v}_{\star}$ in the call-by-value dual calculus and $c = c'$ in the call-by-value dual calculus implies $c^{\star}_{v} = c'^{\star}_{v}$ in the $\mu\tilde{\mu}^{\oplus,\otimes,\neg,\to}_{\mathcal{V}}$-calculus, and similarly for (co-)terms.

The inversion of the translation follows by induction on the syntax of both languages. In each direction, the round-trip translation of the core $\mu\tilde{\mu}$ sublanguage (commands, (co-)variables, and μ- and $\tilde{\mu}$-abstractions), as well as the round-trip translation of injections, pairs, negation co-terms, and call stacks, follows directly by the inductive hypothesis. The other cases for the round-trip translation of the $\mu\tilde{\mu}^{\oplus,\otimes,\neg,\to}_{\mathcal{V}}$-calculus are:

$$\begin{aligned}
\mu(\neg\,[x].c)^{v\star}_{\star v} &=_{IH} \mu(\neg\,[x].\langle x \| \tilde{\mu}x.c\rangle) =_{\tilde{\mu}_{\mathcal{V}}} \mu(\neg\,[x].c) \\
\mu([x \cdot \beta].c)^{v\star}_{\star v} &=_{IH} \mu([x \cdot \beta].\langle \mu\beta.c \| \beta\rangle) =_{\mu_{\mathcal{V}}} \mu([x \cdot \beta].c) \\
\tilde{\mu}[\iota_1\,(x).c_1 \mid \iota_2\,(y).c_2]^{v\star}_{\star v} &=_{IH} \tilde{\mu}[\iota_1\,(x).\langle x \| \tilde{\mu}x.c_1\rangle \mid \iota_2\,(y).\langle y \| \tilde{\mu}y.c_2\rangle] \\
&=_{\tilde{\mu}_{\mathcal{V}}} \tilde{\mu}[\iota_1\,(x).c_1 \mid \iota_2\,(y).c_2] \\
\tilde{\mu}[(x,y).c]^{v\star}_{\star v} &=_{IH} \tilde{\mu}z.\,\langle z \| \tilde{\mu}[(x,\_).\langle x \| \tilde{\mu}x.\,\langle z \| \tilde{\mu}[(\_,y).\langle y \| \tilde{\mu}y.c\rangle]\rangle\rangle]\rangle \\
&=_{\tilde{\mu}_{\mathcal{V}}} \tilde{\mu}z.\,\langle z \| \tilde{\mu}[(x,\_).\langle z \| \tilde{\mu}[(\_,y).c]\rangle]\rangle \\
&=_{\eta^{\otimes}_{\mathcal{V}}} \tilde{\mu}[(x,y).\langle (x,y) \| \tilde{\mu}z.\,\langle z \| \tilde{\mu}[(x,\_).\langle z \| \tilde{\mu}[(\_,y).c]\rangle]\rangle\rangle] \\
&=_{\tilde{\mu}_{\mathcal{V}}} \tilde{\mu}[(x,y).\langle (x,y) \| \tilde{\mu}[(x,\_).\langle (x,y) \| \tilde{\mu}[(\_,y).c]\rangle]\rangle] \\
&=_{\beta^{\otimes}_{\mathcal{S}}} \tilde{\mu}[(x,y).c]
\end{aligned}$$

where the most interesting case is for the round-trip of a case abstraction on a pair, which requires the $\beta\eta$ laws for $\otimes$ to simplify. The other cases for the round-trip translation of the call-by-value dual calculus are:

$$\mathsf{not}(e)^{\star v}_{v\star} =_{IH} \mathsf{not}(\tilde{\mu}x.\,\langle x \| e\rangle) =_{\eta_{\tilde{\mu}}} \mathsf{not}(e)$$

$$\begin{aligned}
\lambda x.v^{\star v}_{v\star} &=_{IH} \lambda x.\mu\beta.\left\langle v \middle\| \beta \right\rangle =_{\eta_\mu} \lambda x.v \\
\pi_1\left[e\right]^{\star v}_{v\star} &=_{IH} \tilde{\mu}z.\left\langle z \middle\| \pi_1\left[\tilde{\mu}x.\left\langle z \middle\| \pi_2\left[\tilde{\mu}y.\left\langle x \middle\| e \right\rangle\right]\right\rangle\right]\right\rangle \\
&=_{\mu_{\mathcal{V}}} \tilde{\mu}z.\left\langle z \middle\| \pi_1\left[\tilde{\mu}x.\left\langle \mu\beta.\left\langle z \middle\| \pi_2\left[\beta\right]\right\rangle \middle\| \tilde{\mu}y.\left\langle x \middle\| e \right\rangle\right\rangle\right]\right\rangle \\
&=_{\tilde{\mu}_{\mathcal{V}'}} \tilde{\mu}z.\left\langle z \middle\| \pi_1\left[\tilde{\mu}x.\left\langle x \middle\| e \right\rangle\right]\right\rangle =_{\eta_{\tilde{\mu}}} \pi_1\left[e\right] \\
\pi_2\left[e\right]^{\star v}_{v\star} &=_{IH} \tilde{\mu}z.\left\langle z \middle\| \pi_1\left[\tilde{\mu}x.\left\langle z \middle\| \pi_2\left[\tilde{\mu}y.\left\langle y \middle\| e \right\rangle\right]\right\rangle\right]\right\rangle \\
&=_{\mu_{\mathcal{V}}} \tilde{\mu}z.\left\langle \mu\alpha.\left\langle z \middle\| \pi_1\left[\alpha\right]\right\rangle \middle\| \tilde{\mu}x.\left\langle z \middle\| \pi_2\left[\tilde{\mu}y.\left\langle y \middle\| e \right\rangle\right]\right\rangle\right\rangle \\
&=_{\tilde{\mu}_{\mathcal{V}'}} \tilde{\mu}z.\left\langle z \middle\| \pi_2\left[\tilde{\mu}y.\left\langle y \middle\| e \right\rangle\right]\right\rangle =_{\eta_{\tilde{\mu}}} \pi_2\left[e\right] \\
\left[e_1, e_2\right]^{\star v}_{v\star} &=_{IH} \left[\tilde{\mu}x.\left\langle x \middle\| e_1 \right\rangle, \tilde{\mu}y.\left\langle y \middle\| e_2 \right\rangle\right] =_{\eta_{\tilde{\mu}}} \left[e_1, e_2\right]
\end{aligned}$$

where the most interesting cases are the π_1 and π_2 projections which requires the extended notion of values in $\mathcal{V}'$ to simplify.

The soundness of equations follows by cases on the possible rewrite rules of the respective equational theories, which may make use of the facts that substitution commutes with translation (since both translations are compositional and hygienic (Downen & Ariola, 2014a)) and $\mathcal{V}$ (co-)values translate to $\mathcal{V}$ (co-)values in both directions. The cases for the core $\mu_{\mathcal{V}}$, $\tilde{\mu}_{\mathcal{V}}$, η_μ, and $\eta_{\tilde{\mu}}$ rules are immediate since they are the same in both calculi. The one tricky issue in relating the core $\mu\tilde{\mu}$ calculus is the extended notion of $\mathcal{V}'$ value, which does *not* translate to a value in $\mu\tilde{\mu}_{\mathcal{V}}$. Thankfully, these extra terms are still semantically substitutable within the $\mu\tilde{\mu}^{\oplus,\otimes,\neg,\to}_{\mathcal{V}}$ equational theory. In particular, we have the following derived equality for $\tilde{\mu}_{\mathcal{V}'}$ within $\mu\tilde{\mu}^{\oplus,\otimes,\neg,\to}_{\mathcal{V}}$ by induction on the values of $\mathcal{V}'$. The case for a first projection value is

$$\begin{aligned}
&\left\langle \mu\alpha.\left\langle V \middle\| \pi_1\left[\alpha\right]\right\rangle \middle\| \tilde{\mu}z.c \right\rangle^\star_v \\
&= \left\langle \mu\alpha.\left\langle V^\star_v \middle\| \tilde{\mu}[(x,y).\langle x \| \alpha \rangle]\right\rangle \middle\| \tilde{\mu}x.c^\star_v \right\rangle \\
&=_{\mu_{\mathcal{V}}} \left\langle V^\star_v \middle\| \tilde{\mu}[(x,y).\langle z \| \tilde{\mu}z.c^\star_v \rangle]\right\rangle \\
&=_{\tilde{\mu}_{\mathcal{V}}} \left\langle V^\star_v \middle\| \tilde{\mu}[(x,y).c^\star_v\left\{x/z\right\}]\right\rangle \\
&=_{\eta_\mu} \left\langle V^\star_v \middle\| \tilde{\mu}[(x,y).c^\star_v\left\{\mu\alpha.\left\langle x \middle\| \alpha \right\rangle/z\right\}]\right\rangle \\
&=_{\beta^\otimes_{\mathcal{V}}} \left\langle V^\star_v \middle\| \tilde{\mu}[(x,y).c^\star_v\left\{\mu\alpha.\left\langle (x,y) \middle\| \tilde{\mu}[(x,y).\langle x \| \alpha \rangle]\right\rangle/z\right\}]\right\rangle \\
&=_{\beta^\otimes_{\mathcal{V}}} \left\langle V^\star_v \middle\| \tilde{\mu}[(x,y).\langle (x,y) \| \tilde{\mu}z'.c^\star_v\left\{\mu\alpha.\left\langle z' \middle\| \tilde{\mu}[(x,y).\langle x \| \alpha \rangle]\right\rangle/z\right\}\rangle]\right\rangle \\
&=_{\eta^\otimes_{\mathcal{V}}} \left\langle V^\star_v \middle\| \tilde{\mu}z'.c^\star_v\left\{\mu\alpha.\left\langle z' \middle\| \tilde{\mu}[(x,y).\langle x \| \alpha \rangle]\right\rangle/z\right\}\right\rangle
\end{aligned}$$

$$
\begin{aligned}
&=_{IH} c^{\star}_{v} \{\mu\alpha.\, \langle V^{\star}_{v} \| \tilde{\mu}[(x,y).\langle x \| \alpha\rangle]\rangle / z\} \\
&= c^{\star}_{v} \{(\mu\alpha.\, \langle V \| \pi_1 [\alpha]\rangle)^{\star}_{v} / z\} = (c\, \{\mu\alpha.\, \langle V \| \pi_1 [\alpha]\rangle / z\})^{\star}_{v}
\end{aligned}
$$

and the case for a second projection value is similar. What remains is to check the soundness of the rewrite rules for each connective. The ς rules are the same in both calculi, so they translate directly. Going from $\mu\tilde{\mu}^{\oplus,\otimes,\neg,\rightarrow}_{\mathcal{V}}$ to the call-by-dual calculus, we have:

$$
\begin{aligned}
(\beta^{\oplus})\quad & \langle \iota_i\,(v) \| \tilde{\mu}[\iota_1\,(x_1).c_1 \mid \iota_2\,(x_2).c_2]\rangle^{v}_{\star} \\
& = \langle \iota_i\,(v^{v}_{\star}) \| [\tilde{\mu}x_1.c_1{}^{v}_{\star}, \tilde{\mu}x_2.c_2{}^{v}_{\star}]\rangle =_{\varsigma^{\oplus}_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}} \langle v^{v}_{\star} \| \tilde{\mu}z.\, \langle \iota_i\,(z) \| [\tilde{\mu}x_1.c_1{}^{v}_{\star}, \tilde{\mu}x_2.c_2{}^{v}_{\star}]\rangle\rangle \\
& =_{\beta^{\oplus}_{\mathcal{V}}} \langle v^{v}_{\star} \| \tilde{\mu}z.\, \langle z \| \tilde{\mu}x_i.c_i{}^{v}_{\star}\rangle\rangle =_{\tilde{\mu}_{\mathcal{V}}} \langle v^{v}_{\star} \| \tilde{\mu}x_i.c_i{}^{v}_{\star}\rangle = \langle v \| \tilde{\mu}x_i.c_i\rangle^{v}_{\star}
\end{aligned}
$$

$$
\begin{aligned}
(\beta^{\otimes})\quad & \langle (v_1, v_2) \| \tilde{\mu}[(x,y).c]\rangle^{v}_{\star} \\
& = \langle (v_1{}^{v}_{\star}, v_2{}^{v}_{\star}) \| \tilde{\mu}z.\, \langle z \| \pi_1\,[\tilde{\mu}x.\, \langle z \| \pi_2\,[\tilde{\mu}y.c^{v}_{\star}]\rangle]\rangle\rangle \\
& =_{\varsigma^{\times}_{\mathcal{V}}\mu_{\mathcal{V}}} \langle v_1{}^{v}_{\star} \| \tilde{\mu}x.\, \langle (x, v_2{}^{v}_{\star}) \| \tilde{\mu}z.\, \langle z \| \pi_1\,[\tilde{\mu}x.\, \langle z \| \pi_2\,[\tilde{\mu}y.c^{v}_{\star}]\rangle]\rangle\rangle\rangle \\
& =_{\varsigma^{\times}_{\mathcal{V}}\mu_{\mathcal{V}}} \langle v_1{}^{v}_{\star} \| \tilde{\mu}x.\, \langle v_2{}^{v}_{\star} \| \tilde{\mu}y.\, \langle (x,y) \| \tilde{\mu}z.\, \langle z \| \pi_1\,[\tilde{\mu}x.\, \langle z \| \pi_2\,[\tilde{\mu}y.c^{v}_{\star}]\rangle]\rangle\rangle\rangle\rangle \\
& =_{\tilde{\mu}_{\mathcal{V}}} \langle v_1{}^{v}_{\star} \| \tilde{\mu}x.\, \langle v_2{}^{v}_{\star} \| \tilde{\mu}y.\, \langle (x,y) \| \pi_1\,[\tilde{\mu}x.\, \langle (x,y) \| \pi_2\,[\tilde{\mu}y.c^{v}_{\star}]\rangle]\rangle\rangle\rangle \\
& =_{\beta^{\times}_{\mathcal{V}}} \langle v_1{}^{v}_{\star} \| \tilde{\mu}x.\, \langle v_2{}^{v}_{\star} \| \tilde{\mu}y.\, \langle x \| \tilde{\mu}x.\, \langle y \| \tilde{\mu}y.c^{v}_{\star}\rangle\rangle\rangle\rangle \\
& =_{\tilde{\mu}_{\mathcal{V}}} \langle v_1{}^{v}_{\star} \| \tilde{\mu}x.\, \langle v_2{}^{v}_{\star} \| \tilde{\mu}y.c^{v}_{\star}\rangle\rangle = \langle v_1 \| \tilde{\mu}x.\, \langle v_2 \| \tilde{\mu}y.c\rangle\rangle^{v}_{\star}
\end{aligned}
$$

$$
(\beta^{\neg})\quad \langle \mu(\neg\,[x].c) \| \neg\,[v]\rangle^{v}_{\star} = \langle \mathsf{not}(\tilde{\mu}x.c^{v}_{\star}) \| \mathsf{not}[v^{v}_{\star}]\rangle =_{\beta^{\neg}_{\mathcal{V}}} \langle v^{v}_{\star} \| \tilde{\mu}x.c^{v}_{\star}\rangle = \langle v \| \tilde{\mu}x.c\rangle^{v}_{\star}
$$

$$
\begin{aligned}
(\beta^{\rightarrow})\quad & \langle \mu([x \cdot \beta].c) \| v \cdot e\rangle^{v}_{\star} = \langle \lambda x.\mu\beta.c^{v}_{\star} \| v^{v}_{\star} \cdot e^{v}_{\star}\rangle =_{\varsigma^{\rightarrow}_{\mathcal{V}}\tilde{\mu}_{\mathcal{V}}} \langle v^{v}_{\star} \| \tilde{\mu}x.\, \langle \lambda x.\mu\beta.c^{v}_{\star} \| x \cdot e^{v}_{\star}\rangle\rangle \\
& =_{\beta^{\rightarrow}_{\mathcal{V}}} \langle v^{v}_{\star} \| \tilde{\mu}x.\, \langle \mu\beta.c^{v}_{\star} \| e^{v}_{\star}\rangle\rangle = \langle v \| \tilde{\mu}x.\, \langle \mu\beta.c \| e\rangle\rangle^{v}_{\star}
\end{aligned}
$$

$$
\begin{aligned}
(\eta^{\oplus})\qquad & \tilde{\mu}[\iota_1\,(x).\langle \iota_1\,(x) \| \alpha\rangle \mid \iota_2\,(y).\langle \iota_2\,(y) \| \alpha\rangle]^{v}_{\star} \\
& = [\tilde{\mu}x.\, \langle \iota_1\,(x) \| \alpha\rangle, \tilde{\mu}y.\, \langle \iota_2\,(y) \| \alpha\rangle] =_{\eta^{+}_{\mathcal{V}}} \alpha
\end{aligned}
$$

$$
\begin{aligned}
(\eta^{\otimes})\qquad & \tilde{\mu}[(x,y).\langle (x,y) \| \alpha\rangle]^{v}_{\star} \\
& = \tilde{\mu}z.\, \langle z \| \pi_1\,[\tilde{\mu}x.\, \langle z \| \pi_2\,[\tilde{\mu}y.\, \langle (x,y) \| \alpha\rangle]\rangle]\rangle \\
& =_{\eta^{\times}_{\mathcal{V}}\beta^{\times}_{\mathcal{V}'}} \tilde{\mu}z.\, \langle \mu\beta_1.\, \langle z \| \pi_1\,[\beta_1]\rangle \| \tilde{\mu}x.\, \langle z \| \pi_2\,[\tilde{\mu}y.\, \langle (x,y) \| \alpha\rangle]\rangle\rangle \\
& =_{\tilde{\mu}_{\mathcal{V}'}} \tilde{\mu}z.\, \langle z \| \pi_2\,[\tilde{\mu}y.\, \langle (\mu\beta_1.\, \langle z \| \pi_1\,[\beta_1]\rangle, y) \| \alpha\rangle]\rangle \\
& =_{\eta^{\times}_{\mathcal{V}}\beta^{\times}_{\mathcal{V}'}} \tilde{\mu}z.\, \langle \mu\beta_2.\, \langle z \| \pi_2\,[\beta_2]\rangle \| \tilde{\mu}y.\, \langle (\mu\beta_1.\, \langle z \| \pi_1\,[\beta_1]\rangle, y) \| \alpha\rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
&=_{\tilde{\mu}_{\mathcal{V}'}} \tilde{\mu} z.\, \langle (\mu\beta_1.\, \langle z \| \pi_1 [\beta_1] \rangle, \mu\beta_2.\, \langle z \| \pi_2 [\beta_2] \rangle) \| \alpha \rangle \\
&=_{\eta_{\mathcal{V}}^{\times}} \tilde{\mu} z.\, \langle z \| \alpha \rangle =_{\eta_{\tilde{\mu}}} \alpha
\end{aligned}
$$

$$
(\eta^{\neg}) \qquad \mu(\neg [x].\langle z \| \neg [x] \rangle)^{v}_{\star} = \mathsf{not}(\tilde{\mu} x.\, \langle z \| \mathsf{not}[x] \rangle) =_{\eta_{\mathcal{V}}^{\neg}} z
$$

$$
(\eta^{\rightarrow}) \qquad \mu([x \cdot \beta].\langle z \| x \cdot \beta \rangle)^{v}_{\star} = \lambda x.\mu\beta.\, \langle z \| x \cdot \beta \rangle =_{\eta_{\mathcal{V}}^{\rightarrow}} z
$$

Going from the call-by-value dual calculus to $\mu\tilde{\mu}_{\mathcal{V}}^{\oplus,\otimes,\neg,\rightarrow}$, we have:

$$
\begin{aligned}
(\beta_{\mathcal{V}}^{+}) \qquad \langle \iota_i (V) \| [e_1, e_2] \rangle^{\star}_{v} &= \langle \iota_i (V^{\star}_{v}) \| \tilde{\mu}[\iota_1 (x).\langle x \| e_{1}{}^{\star}_{v} \rangle \mid \iota_2 (x).\langle x \| e_{2}{}^{\star}_{v} \rangle] \rangle \\
&=_{\beta_{\mathcal{V}}^{\oplus}} \langle V^{\star}_{v} \| e_{i}{}^{\star}_{v} \rangle = \langle V \| e_i \rangle^{\star}_{v}
\end{aligned}
$$

$$
\begin{aligned}
(\beta_{\mathcal{V}}^{\times}) \qquad \langle (V_1, V_2) \| \pi_i [e] \rangle^{\star}_{v} &= \langle (V_{1}{}^{\star}_{v}, V_{2}{}^{\star}_{v}) \| \tilde{\mu}[(x_1, x_2).\langle x_i \| e^{\star}_{v} \rangle] \rangle \\
&=_{\beta_{\mathcal{V}}^{\otimes}} \langle V_{i}{}^{\star}_{v} \| e^{\star}_{v} \rangle = \langle V_i \| e \rangle^{\star}_{v}
\end{aligned}
$$

$$
\begin{aligned}
(\beta_{\mathcal{V}}^{\neg}) \qquad \langle \mathsf{not}(e) \| \mathsf{not}(v) \rangle^{\star}_{v} &= \langle \mu(\neg [x].\langle x \| e^{\star}_{v} \rangle) \| \neg [v^{\star}_{v}] \rangle \\
&=_{\beta_{\mathcal{V}}^{\neg}} \langle v^{\star}_{v} \| e^{\star}_{v} \rangle = \langle v \| e \rangle^{\star}_{v}
\end{aligned}
$$

$$
\begin{aligned}
(\beta_{\mathcal{V}}^{\rightarrow}) \qquad \langle \lambda x.v \| V \cdot e \rangle^{\star}_{v} &= \langle \mu([x \cdot \beta].\langle v^{\star}_{v} \| \beta \rangle) \| V^{\star}_{v} \cdot e^{\star}_{v} \rangle \\
&=_{\beta^{\rightarrow}\mu_{\mathcal{V}}} \langle V^{\star}_{v} \| \tilde{\mu} x.\, \langle v^{\star}_{v} \| e^{\star}_{v} \rangle \rangle \\
&=_{\tilde{\mu}_{\mathcal{V}'}} \langle v^{\star}_{v} \{ V^{\star}_{v} / x \} \| e^{\star}_{v} \rangle = \langle v \{ V / x \} \| e \rangle^{\star}_{v}
\end{aligned}
$$

$$
\begin{aligned}
(\eta_{\mathcal{V}}^{+}) \quad & [\tilde{\mu} x.\, \langle \iota_1 (x) \| e \rangle, \tilde{\mu} y.\, \langle \iota_2 (y) \| e \rangle]^{\star}_{v} \\
& = \tilde{\mu}[\iota_1 (x).\langle x \| \tilde{\mu} x.\, \langle \iota_1 (x) \| e^{\star}_{v} \rangle \rangle \mid \iota_2 (y).\langle y \| \tilde{\mu} y.\, \langle \iota_2 (y) \| e^{\star}_{v} \rangle \rangle] \\
& =_{\tilde{\mu}_{\mathcal{V}}} \tilde{\mu}[\iota_1 (x).\langle \iota_1 (x) \| e^{\star}_{v} \rangle \mid \iota_2 (y).\langle \iota_2 (y) \| e^{\star}_{v} \rangle] =_{\eta_{\mathcal{V}}^{\oplus}} e^{\star}_{v}
\end{aligned}
$$

$$
\begin{aligned}
(\eta_{\mathcal{V}}^{\times}) \quad & (\mu\alpha.\, \langle v \| \pi_1 [\alpha] \rangle, \mu\beta.\, \langle v \| \pi_2 [\beta] \rangle)^{\star}_{v} \\
& = (\mu\alpha.\, \langle V^{\star}_{v} \| \tilde{\mu}[(x, \_).\langle x \| \alpha \rangle] \rangle, \mu\beta.\, \langle V^{\star}_{v} \| \tilde{\mu}[(\_, y).\langle y \| \beta \rangle] \rangle) \\
& =_{\eta_{\mu}} \mu\gamma.\, \langle (\mu\alpha.\, \langle V^{\star}_{v} \| \tilde{\mu}[(x, \_).\langle x \| \alpha \rangle] \rangle, \mu\beta.\, \langle V^{\star}_{v} \| \tilde{\mu}[(\_, y).\langle y \| \beta \rangle] \rangle) \| \gamma \rangle \\
& =_{\tilde{\mu}_{\mathcal{V}'}} \mu\gamma.\, \langle V^{\star}_{v} \| \tilde{\mu} z.\, \langle (\mu\alpha.\, \langle z \| \tilde{\mu}[(x, \_).\langle x \| \alpha \rangle] \rangle, \mu\beta.\, \langle z \| \tilde{\mu}[(\_, y).\langle y \| \beta \rangle] \rangle) \| \gamma \rangle \rangle \\
& =_{\eta_{\mathcal{V}}^{\otimes}} \mu\gamma.\, \left\langle V^{\star}_{v} \middle\| \tilde{\mu} \left[(x, y).\left\langle (x, y) \middle\| \tilde{\mu} z.\, \left\langle \begin{matrix} (\mu\alpha.\, \langle z \| \tilde{\mu}[(x, \_).\langle x \| \alpha \rangle] \rangle, \\ \mu\beta.\, \langle z \| \tilde{\mu}[(\_, y).\langle y \| \beta \rangle] \rangle) \end{matrix} \middle\| \gamma \right\rangle \right\rangle \right] \right\rangle \\
& =_{\tilde{\mu}_{\mathcal{V}}} \mu\gamma.\, \left\langle V^{\star}_{v} \middle\| \tilde{\mu} \left[(x, y).\left\langle \begin{matrix} (\mu\alpha.\, \langle (x, y) \| \tilde{\mu}[(x, \_).\langle x \| \alpha \rangle] \rangle, \\ \mu\beta.\, \langle (x, y) \| \tilde{\mu}[(\_, y).\langle y \| \beta \rangle] \rangle) \end{matrix} \middle\| \gamma \right\rangle \right] \right\rangle
\end{aligned}
$$

$$
\begin{aligned}
&=_{\beta_{\mathcal{V}}^{\otimes}} \mu\gamma.\, \langle V_v^{\star} \| \tilde{\mu}[(x,y).\langle(\mu\alpha.\,\langle x\|\alpha\rangle, \mu\beta.\,\langle y\|\beta\rangle)\|\gamma\rangle]\rangle \\
&=_{\eta_\mu} \mu\gamma.\, \langle V_v^{\star} \| \tilde{\mu}[(x,y).\langle(x,y)\|\gamma\rangle]\rangle \\
&=_{\eta^{\otimes}} \mu\gamma.\, \langle V_v^{\star}\|\gamma\rangle =_{\eta_\mu} V_v^{\star}
\end{aligned}
$$

$$
\begin{aligned}
(\eta_{\mathcal{V}}^{\neg})\quad & \mathsf{not}(\tilde{\mu}x.\,\langle V\|\mathsf{not}[x]\rangle)_v^{\star} = \mu(\neg\,[x].\langle x\|\tilde{\mu}x.\,\langle V_v^{\star}\|\neg\,[x]\rangle\rangle) \\
& =_{\tilde{\mu}_{\mathcal{V}}} \mu(\neg\,[x].\langle V_v^{\star}\|\neg\,[x]\rangle) =_{\eta_{\mathcal{V}'}^{\neg}} V_v^{\star}
\end{aligned}
$$

$$
\begin{aligned}
(\eta_{\mathcal{V}}^{\to})\quad & \lambda x.\mu\beta.\,\langle V\|x\cdot\beta\rangle_v^{\star} = \mu([x\cdot\beta].\mu\beta.\,\langle V_v^{\star}\|x\cdot\beta\rangle\beta) \\
& =_{\tilde{\mu}_{\mathcal{V}}} \mu([x\cdot\beta].\langle V_v^{\star}\|x\cdot\beta\rangle) =_{\eta_{\mathcal{V}'}^{\to}} V_v^{\star}
\end{aligned}
$$

b) This follows from part (a) by duality. More specifically, translation commutes with duality in the two calculi as

$$
c_v^{\star\perp} = {c^{\perp}}^{\star}_{n} \qquad c_n^{\star\perp} = {c^{\perp}}^{\star}_{v} \qquad c_{\star}^{v\perp} = {c^{\perp}}^{n}_{\star} \qquad c_{\star}^{n\perp} = {c^{\perp}}^{v}_{\star}
$$

and similarly for (co-)terms, which follows directly from the definitions of the duality and translation operations by (mutual) induction on the syntax of commands and (co-)terms. Therefore, the fact that the translations are inverses comes from part (a) by applying Theorems 3.6 and 5.5, so

$$
\begin{aligned}
c_{\star n}^{n\star} &=_{Theorem\ 5.5} {c_{\star n}^{n\star}}^{\perp\perp} = {c^{\perp\perp}}^{v\star}_{\star v} = c^{\perp\perp} =_{Theorem\ 5.5} c \\
c_{n\star}^{\star n} &=_{Theorem\ 3.6} {c_{n\star}^{\star n}}^{\perp\perp} = {c^{\perp\perp}}^{\star v}_{v\star} = c^{\perp\perp} =_{Theorem\ 3.6} c
\end{aligned}
$$

and similarly for (co-)terms. Furthermore $\mathcal{N}$ is dual to $\mathcal{V}$ and $\&, \invamp, \sim, \to$ is dual to $\oplus, \otimes, \neg, \to$, so if we have $c = c'$ in the $\mu\tilde{\mu}_{\mathcal{N}}^{\&,\invamp,\sim,\to}$ then $c^{\perp} = c^{\perp}$ in $\mu\tilde{\mu}_{\mathcal{N}}^{\oplus,\otimes,\neg,\to}$ by Theorem 5.7, ${c^{\perp}}^{v}_{\star} = {c'^{\perp}}^{v}_{\star}$ in the call-by-value dual calculus by part (a), and thus

$$
c_{\star}^{n} = {c_{\star}^{n}}^{\perp\perp} = {{c^{\perp}}^{v}_{\star}}^{\perp} = {{c'^{\perp}}^{v}_{\star}}^{\perp} = {c'^{n}_{\star}}^{\perp\perp} = c_{\star}^{n}
$$

in the call-by-name dual calculus by Theorems 3.8 and 3.6 and the above, and similarly for (co-)terms. Going the other way, if we have $c = c'$ in the call-by-value dual calculus then we have ${c^{\perp}}^{\star}_{v} = {c'^{\perp}}^{\star}_{v}$ in $\mu\tilde{\mu}_{\mathcal{V}}^{\oplus,\otimes,\neg,\to}$ by Theorem 5.7 and part (a), so

$$c_n^\star = {c_n^\star}^{\perp\perp} = {c^\perp}_v^{\star\perp} = {c'^\perp}_v^{\star\perp} = {c'^\star_n}^{\perp\perp} = c_n^\star$$

in $\mu\tilde{\mu}_{\mathcal{N}}^{\&,⅋,\sim,\to}$ by Theorem 3.8 and the above, and similarly for (co-)terms. □

It follows that the idea of distinguishing data and co-data provides a unifying framework for studying the computational meaning of types in the sequent calculus. The distinction is baked into polarized languages, like system L as previously seen in Chapter IV. But even for the dual calculi, in which there is no apparent division between data types and co-data types, the difference between the two is instead buried inside the dual call-by-value and call-by-name interpretations of the types. Next in the following Chapter VI, we will move beyond just the simple types considered here (variations of products, sums, functions, and so on) to also incorporate more advanced type features into the data and co-data framework. In particular, Chapter VI will show how polymorphism in the form of type abstraction, previously seen in Chapters II and III, can be rephrased in terms of the data and co-data framework explored here. This extension will serve as a platform to study the duality between *induction* and *co-induction* as two modes of *structural recursion* which improves the treatment of co-induction as the equal-and-opposite partner to induction, and also clarifies the murky issues of "well-foundedness" surrounding co-induction. In particular, the tendency to view co-inductive objects as "necessarily lazy" comes from the fact that they are co-data objects. The delicate balance of evaluation order that is required to combine inductive and co-inductive objects falls out automatically by modeling as data and co-data which already implies the correct computational meaning.

CHAPTER VI

Induction and Co-Induction

Martin-Löf's type theory (Martin-Löf, 1998, 1975; Martin-Löf, 1982) taught us that inductive definitions and reasoning are pervasive throughout proof theory, mathematics, and computer science. Inductive data types are used in programming languages like ML and Haskell to represent structures, and in proof assistants and dependently typed languages like Coq and Agda to reason about finite structures of arbitrary size. Mendler (1988) showed us how to talk about recursive types and formalize inductive reasoning over arbitrary data structures. However, the foundation for the opposite to induction, co-induction, has not fared so well. Co-induction is a major concept in programming, representing endless processes, but it is often neglected, misunderstood, or mistreated. As articulated by McBride (Singh *et al.*, 2011):

> We are obsessed with foundations partly because we are aware of a number of significant foundational problems that we've got to get right before we can do anything realistic. The thing I would think of . . . in particular in that respect is co-induction and reasoning about co-recursive processes. That's currently, in all major implementations of type theory, a disaster. And if we're going to talk about real systems, we've got to actually have something sensible to say about that.

The introduction of co-patterns for co-induction Abel *et al.* (2013) is a major step forward in rectifying this situation. Abel et al. emphasize that there is a dual view to inductive data types, in which the values of types are defined by how they are used instead of how they are built, a perspective on *co-data types* first spurred on by Hagino (1987, 1989). Co-inductive co-data types are exciting because they may solve the existing problems with representing infinite objects in proof assistants like Coq (Abel & Pientka, 2013).

Our goal here is to improve the understanding and treatment of co-induction, and to integrate both induction and co-induction into a cohesive whole for representing well-founded recursive programs. Our main tools for accomplishing this goal are the pervasive and overt duality and symmetry that runs through classical logic and

This chapter is a revised version of (Downen *et al.*, 2015) to fit in the context of this dissertation.

the sequent calculus. By developing a representation of well-founded induction in a language for the classical sequent calculus, we get an equal and opposite version of well-founded co-induction “for free.” Thus, the challenges that arise from using classical sequent calculus as a foundation for induction are just as well the challenges of co-induction, as the two are inherently developed simultaneously. Later in Chapter IX, we will translate the developments of induction and co-induction in the classical sequent calculus to a λ-calculus based language for effect-free programs, to better relate to the current practice of type theory and functional programming. As the λ-based style lacks symmetries present in the sequent calculus, some of the constructs for recursion are lost in translation. Unsurprisingly, the cost of an asymmetrical viewpoint is blindness to the complete picture revealed by duality.

Our philosophy is to emphasize the disentanglement of the recursion in types from the recursion in programs, to attain a language rich in both data and co-data while highlighting their dual symmetries. On the one hand, the Coq viewpoint is that *all* recursive types—both inductive and co-inductive—are represented as data types (positive types in polarized logic (Munch-Maccagnoni, 2009)), where induction allows for infinitely deep destruction and co-induction allows for infinitely deep construction. On the other hand, the co-pattern approach (Abel *et al.*, 2013; Abel & Pientka, 2013) which is inspired by Hagino’s (1987) treatment of co-induction via finiate observations represents inductive types as data and co-inductive types as co-data. In contrast, we take the view that separates the recursive definition of types from the types used for specifying recursive processing loops. Thereby, the types for representing the structure of a recursive process are given first-class status, defined on their own independently of any other programming construct. This makes the types more compositional, so that they may be combined freely in more ways, as they are not confined to certain restrictions about how they relate to data vs co-data or induction vs co-induction. More traditional views on the distinction between inductive and co-inductive programs come from different modes of use for the same building blocks, emerging from particular compositions of several (co-)data types.

We will base our calculus for recursion on the parametric $\mu\tilde{\mu}$-calculus with data and co-data from Chapter V which corresponds to a classical logic, so it inherently contains *control effects* (Griffin, 1990) that allow programs to abstract over their own control-flow—intuitionistic logic and effect-free functional programs are later considered as a special case in Chapter IX. As we saw, the fundamental dilemma of classical computation (Section 3.2) means that the intended evaluation strategy for a program becomes an essential part of understanding its meaning: even terminating programs give different results for different strategies. For example, the functional program $\mathit{length}(\mathsf{Cons}\ (\mathit{error}\ \text{“}\mathtt{boom}\text{”})\ \mathsf{Nil})$ returns 1 under call-by-name (lazy) evaluation, but goes “boom” with an error under call-by-value (strict) evaluation. Therefore, a calculus that talks about the behavior of programs needs to consider the impact of the evaluation strategy. We therefore leverage the parametric nature of the $\mu\tilde{\mu}$-calculus to disentangle this choice from the calculus itself, boiling down the distinction as a substitution strategy. Note that, unlike many accounts of co-induction, we do not rely on a particular choice of evaluation strategy—like some sort of lazy evaluation which delays computing results until they are needed—but instead the apt use of data and co-data forces the correct interpretation of infinite objects. We therefore get a family of calculi, parameterized by the strategy, for reasoning about the behavior of programs ultimately executed with some evaluation strategy. The issue of strong normalization is then framed uniformly over this family of calculi by specifying some basic requirements of the chosen substitution strategy which are inspired by focusing in logic.

The bedrock on which we build our structures for recursion is the connection between logic and programming languages, and the cornerstone of the design is the duality permeating these programming concepts. Induction and co-induction are clearly dual, and the duality of their opposition shines through in the the symmetric setting of the sequent calculus. Here, classicality is not just a feature, but an essential completion of the duality needed to fully express the connections between recursion and co-recursion. We consider several different types for representing recursion in programs based on the mathematical principles of *primitive* and *noetherian* recursion which are reflected as pairs of dual data and co-data types. As we will find, both of these different recursive principles have different strengths as programming features: primitive recursion allows us to depend on the statically-known sizes of constructions at run-time à la GADTs and simulate seemingly infinite constructed objects, like potentially infinite lists in Coq or Haskell, whereas noetherian recursion admits type-erasure. In essence, we demonstrate how this parametric sequent calculus can be used as a core calculus and compilation target for establishing well-foundedness of recursive programs, via the computational interpretation of common principles of mathematical induction.

This chapter covers the following topics:

- A presentation of some basic functional programs, including co-patterns (Abel *et al.*, 2013), in a sequent based syntax to illustrate how the sequent calculus gives a language for programming with structures and duality (Section 6.1).
- A language for the higher-order sequent calculus in which *all* types, including functions and polymorphism, are treated as user-defined data and co-data types (Section 6.2).
- Two forms of well-founded recursion in types—based on primitive and noetherian recursion—along with specific data and co-data types for performing well-founded recursion in programs (Section 6.3).
- An extension of the language of the sequent calculus with recursion, where the reduction theory is strongly normalizing for well-typed programs and supports erasure of computationally irrelevant types at run-time (Section 6.4).

Programming with Structures and Duality

Pattern-matching is an integral part of functional programming languages, and is a great boon to their elegance. However, the traditional language of pattern-matching can be lacking in areas, especially when we consider *dual* concepts that arise in all programs. For example, when defining a function by patterns, we can match on the structure of the *input*—the argument given to the function—but not its *output*—the observation being made about its result. In contrast, calculi inspired by the sequent calculus that we've seen in Chapters III, IV, and V feature a more symmetric language which both highlights and restores this missing duality. Indeed, in a setting with such ingrained symmetry, maintaining dualities is natural. We now consider how concepts from functional programming translate to a sequent-based language, and how programs can leverage duality by writing basic recursive functional programs in this symmetric setting.

Example 6.1. One of the most basic functional programs is the function that calculates the length of a list. We can write this *length* function in a Haskell- or Agda-like language by pattern-matching over the structure of the given $\mathsf{List}\ \ a$ to produce a Nat:

$$\begin{array}{ll}
\textbf{data}\,\mathsf{Nat}\,\textbf{where} & \textbf{data}\,\mathsf{List}\,a\,\textbf{where} \\
\quad \mathsf{Z} : \mathsf{Nat} & \qquad \mathsf{Nil} : \mathsf{List}\,a \\
\quad \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat} & \qquad \mathsf{Cons} : a \to \mathsf{List}\,a \to \mathsf{List}\,a
\end{array}$$

$$\begin{array}{ll}
\mathit{length} & : \mathsf{List}\,a \to \mathsf{Nat} \\
\mathit{length}\ \ \mathsf{Nil} & = \mathsf{Z} \\
\mathit{length}\ \ (\mathsf{Cons}\,x\ \ \mathit{xs}) & = \textbf{let}\,y = \mathit{length}\ \ \mathit{xs}\,\textbf{in}\ \mathsf{S}\,y
\end{array}$$

This definition of *length* describes its result for every possible call. Similarly, we can define *length* in the parametric $\mu\tilde{\mu}$-calculus[1] from Chapter V in much the same way. First, we introduce the types in question by data declarations in the sequent calculus:

$$\begin{array}{lrl}
\textbf{data}\,\mathsf{Nat}\,\textbf{where} & & \textbf{data}\,\mathsf{List}(X)\,\textbf{where} \\
\quad \mathsf{Z} : \quad\ \ \vdash \mathsf{Nat} \mid & & \quad \mathsf{Nil} : \qquad\qquad \vdash \mathsf{List}(X) \mid \\
\quad \mathsf{S} : \ \ \mathsf{Nat} \vdash \mathsf{Nat} \mid & & \quad \mathsf{Cons} : \ \ X, \mathsf{List}(X) \vdash \mathsf{List}(X) \mid
\end{array}$$

While these declarations give the same information as before, the differences between these specific data type declarations are largely stylistic. Instead of describing the constructors in terms of a pre-defined function type, the shape of the constructors are described via sequents, replacing function arrows with entailment ($\vdash$) and commas for separating multiple inputs. Furthermore, the type of the main output produced by each constructor is highlighted to the right of the sequent between entailment and a vertical bar, as in $\vdash \mathsf{Nat} \mid$ or $\vdash \mathsf{List}(X) \mid$, and all other types describe the parameters that must be given to the constructor to produce this output. Thus, we can construct a list as either Nil or $\mathsf{Cons}(x, \mathit{xs})$, much like in functional languages. Next, we define *length* by specifying its behavior for every possible call:

$$\begin{array}{ll}
\mathit{length} & : \mathsf{List}(X) \to \mathsf{Nat} \\
\langle \mathit{length} \| \mathsf{Nil} \cdot \alpha \rangle & = \langle \mathsf{Z} \| \alpha \rangle \\
\langle \mathit{length} \| \mathsf{Cons}(x, \mathit{xs}) \cdot \alpha \rangle & = \langle \mathit{length} \| \mathit{xs} \cdot \tilde{\mu} y.\, \langle \mathsf{S}(y) \| \alpha \rangle \rangle
\end{array}$$

[1] Recall that following the notation of Chapter III the symbols μ and $\tilde{\mu}$ used here are not related to recursion, as they sometimes are in other languages, but rather are binders for variables and co-variables.

The main difference is that we consider more than just the argument to *length*. Instead, we are describing the action of *length* with its entire context by showing the behavior of a *command* connecting it together with a consumer. For example, in the command $\langle \mathsf{Z} \| \alpha \rangle$, Z is a *term* producing zero and α is a *co-term*—specifically a *co-variable*—that consumes that number. Besides co-variables, we have other co-terms that consume information. The call-stack $\mathsf{Nil} \cdot \alpha$ consumes a function by supplying it with Nil as its argument and consuming its returned result with α. The input abstraction $\tilde{\mu} y. \langle \mathsf{S}(y) \| \alpha \rangle$ names its input y before running the command $\langle \mathsf{S}(y) \| \alpha \rangle$, similarly to the context $\mathbf{let}\, y = \square\, \mathbf{in}\ \mathsf{S}(y)$ from the functional program.

In functional programs, it is common to avoid explicitly naming the result of a recursive call, especially in such a short program. Instead, we would more likely define *length* as:

$$\begin{aligned}
&\mathit{length} && : \mathsf{List}\, a \to \mathsf{Nat} \\
&\mathit{length}\ \mathsf{Nil} && = \mathsf{Z} \\
&\mathit{length}\ (\mathsf{Cons}\, x\ \mathit{xs}) && = \mathsf{S}\ (\mathit{length}\ \mathit{xs})
\end{aligned}$$

We can mimic this definition in the sequent calculus as:

$$\begin{aligned}
&\mathit{length} && : \mathsf{List}(X) \to \mathsf{Nat} \\
&\langle \mathit{length} \| \mathsf{Nil} \cdot \alpha \rangle && = \langle \mathsf{Z} \| \alpha \rangle \\
&\langle \mathit{length} \| \mathsf{Cons}(x, \mathit{xs}) \cdot \alpha \rangle && = \langle \mathsf{S}(\mu\beta. \langle \mathit{length} \| \mathit{xs} \cdot \beta \rangle) \| \alpha \rangle
\end{aligned}$$

Note that to represent the functional call *length xs* inside the successor constructor S, we need to make use of the output abstraction $\mu\beta. \langle \mathit{length} \| \mathit{xs} \cdot \beta \rangle$ that names its output channel β before running the command $\langle \mathit{length} \| \mathit{xs} \cdot \beta \rangle$, which calls *length* with xs as the argument and β as the return point. As we saw in Section 5.6, output abstractions are exactly dual to input abstractions, and defining *length* in $\mu\tilde{\mu}$ requires us to name the recursive result as either an input or an output.

Just as functions can be represented as first-class values through λ-abstractions in functional languages, their sequent calculus counter-parts can be represented as first-class values in terms of case abstractions in the $\mu\tilde{\mu}$-calculus. Using a recursively-defined case abstraction with deep pattern-matching, we can represent *length* in the

$\mu\tilde{\mu}$-calculus from Chapter 5.2:

$$\begin{aligned} \mathit{length} = \mu(&\mathsf{Nil} \cdot \alpha.\langle \mathsf{Z} \| \alpha \rangle \\ &|\mathsf{Cons}(x, \mathit{xs}) \cdot \alpha.\langle \mathit{length} \| \mathit{xs} \cdot \tilde{\mu}y.\, \langle \mathsf{S}(y) \| \alpha \rangle \rangle) \end{aligned}$$

Furthermore, the deep pattern-matching can be mechanically translated to the shallow case analysis on (co-)data structures:

$$\begin{aligned} \mathit{length} = \mu(\mathit{xs} \cdot \alpha.\, \langle \mathit{xs} \| \tilde{\mu}[&\mathsf{Nil}.\langle \mathsf{Z} \| \alpha \rangle \\ &|\mathsf{Cons}(x, \mathit{xs}').\langle \mathit{length} \| \mathit{xs}' \cdot \tilde{\mu}y.\, \langle \mathsf{S}(y) \| \alpha \rangle \rangle]\rangle) \end{aligned}$$

This case abstraction describes exactly the same specification as the definition for *length* according to the reduction theory of the parametric $\mu\tilde{\mu}$-calculus: when run with the call-stack $\mathsf{Nil} \cdot \alpha$, the command reduces to $\langle \mathsf{Z} \| \alpha \rangle$, and when run with the call-stack $\mathsf{Cons}(x, \mathit{xs}) \cdot \alpha$, the command reduces to $\langle \mathit{length} \| \mathit{xs} \cdot \tilde{\mu}y.\, \langle \mathsf{S}(y) \| \alpha \rangle \rangle$. However, here we will favor presenting the example programs in the style of specifying the behavior of commands using deep pattern-matching, as this gives a higher-level and more abstract reading of programs, with the understanding that they can be mechanically compiled down to (recursive) case abstractions with shallow pattern-matching as above. *End example* 6.1.

We have seen how to write a recursive function by pattern-matching on the first argument, x, in a call-stack $x \cdot \alpha$. However, why should we be limited to only matching on the structure of the argument x? If the observations on the returned result must also follow a particular structure, why can't we match on α as well? Indeed, in a symmetric language, there is no such distinction. For example, the function call-stack itself can be viewed as a structure, so that a curried chain of function applications $f\ x\ y\ z$ is represented by the pattern $x \cdot y \cdot z \cdot \alpha$, which reveals the nested structure down the output side of function application, rather than the input side. Thus, the sequent calculus reveals a dual way of thinking about information in programs phrased as *co-data*, as we saw in Chapter V, in which observations follow predictable patterns, and values respond to those observations by matching on their structure. In such a symmetric setting, it is only natural to match on any structure appearing in *either* inputs or outputs.

Example 6.2. We can consider this view on co-data to understand programs with "infinite" objects. For example, infinite streams may be defined by the primitive projections *out* of streams:

$$
\begin{aligned}
&\textbf{codata}\,\mathsf{Stream}(X)\,\textbf{where}\\
&\quad \mathsf{Head}: \ \mid \mathsf{Stream}(X) \vdash a\\
&\quad \mathsf{Tail}: \ \mid \mathsf{Stream}(X) \vdash \mathsf{Stream}(X)
\end{aligned}
$$

Contrarily to data types, the type of the main input consumed by co-data constructors is highlighted to the left of the sequent in between a vertical bar and entailment, as in $\mid \mathsf{Stream}(X) \vdash$. The rest of the types describe the parameters that must be given to the constructor in order to properly consume this main input. For Streams, the observation $\mathsf{Head}[\alpha]$ requests the head value of a stream which should be given to α, and $\mathsf{Tail}[\beta]$ asks for the tail of the stream which should be given to β.[2] We can now define a function *countUp*—which turns an x of type Nat into the infinite stream $x, \mathsf{S}(x), \mathsf{S}(\mathsf{S}(x)), \ldots$—by pattern-matching on the structure of observations on functions and streams:

$$
\begin{aligned}
&\mathit{countUp} &&: \mathsf{Nat} \to \mathsf{Stream}(\mathsf{Nat})\\
&\langle \mathit{countUp} \| x \cdot \mathsf{Head}[\alpha] \rangle &&= \langle x \| \alpha \rangle\\
&\langle \mathit{countUp} \| x \cdot \mathsf{Tail}[\beta] \rangle &&= \langle \mathit{countUp} \| \mathsf{S}(x) \cdot \beta \rangle
\end{aligned}
$$

If we compare *countUp* with *length* in this style, we can see that there is no fundamental distinction between them: they are both defined by cases on their possible observations. The only point of difference is that *length* happens to match on the structure of its argument in its call-stack, whereas *countUp* matches on the return co-data structure of in its call-stack.

Abel *et al.* (2013) have carried this intuition back into the functional paradigm. For example, we can still describe streams by their Head and Tail projections, and

[2] Keeping the convention from Chapter III, we use square brackets as grouping delimiters in observations, like the head projection $\mathsf{Head}[\alpha]$ out of a stream, as opposed to round parentheses used as grouping delimiters in results, like the successor number $\mathsf{S}(y)$. This helps to disambiguate between results (terms) and observations (co-terms) in a way that is syntactically apparent independently of their context.

define *countUp* through co-patterns:

$$\begin{aligned}&\textbf{codata}\,\mathsf{Stream}\,a\,\textbf{where}\\&\qquad\mathsf{Head} : \mathsf{Stream}\,a \to a\\&\qquad\ \ \mathsf{Tail} : \mathsf{Stream}\,a \to \mathsf{Stream}\,a\end{aligned}$$

$$\begin{aligned}&\mathit{countUp} && : \mathsf{Nat} \to \mathsf{Stream}(X)\\&(\mathit{countUp}\ x).\,\mathsf{Head} && = x\\&(\mathit{countUp}\ x).\,\mathsf{Tail} && = \mathit{countUp}\ (\mathsf{S}\,x)\end{aligned}$$

This definition gives the functional program corresponding to the sequent version of *countUp*. So we can see that co-patterns arise naturally, in Curry-Howard isomorphism style, from the computational interpretation of Gentzen's (1935a) sequent calculus. *End example* 6.2.

Example 6.3. Since a symmetric language is not biased against pattern-matching on inputs or outputs, and indeed the two are treated identically, there is nothing special about matching against *both* inputs and outputs simultaneously. For example, we can model infinite streams with possibly missing elements as

$$\mathsf{SkipStream}(X) = \mathsf{Stream}(\mathsf{Maybe}(X))$$

where $\mathsf{Maybe}(X)$ corresponds to the Haskell data type of the same name defined as:

$$\begin{aligned}&\textbf{data}\,\mathsf{Maybe}(X)\,\textbf{where}\\&\qquad\mathsf{Nothing} : \quad\ \vdash \mathsf{Maybe}(X) \mid\\&\qquad\quad\ \ \mathsf{Just} : X \vdash \mathsf{Maybe}(X) \mid\end{aligned}$$

with constructors $\mathsf{Nothing}$ and $\mathsf{Just}(x)$ for x of type X. Then we can define the *empty* skip stream which gives $\mathsf{Nothing}$ at every position, and the *countDown* function that transforms $\mathsf{S}^n(\mathsf{Z})$ into the stream $\mathsf{S}^n(\mathsf{Z}), \mathsf{S}^{n-1}(\mathsf{Z}), \dots, \mathsf{Z}, \mathsf{Nothing}, \dots$:

$$\begin{aligned}
&\mathit{empty} && : \mathsf{SkipStream}(\mathsf{Nat}) \\
&\langle \mathit{empty} \| \mathsf{Head}[\alpha] \rangle && = \langle \mathsf{Nothing} \| \alpha \rangle \\
&\langle \mathit{empty} \| \mathsf{Tail}[\beta] \rangle && = \langle \mathit{empty} \| \beta \rangle \\
&\mathit{countDown} && : \mathsf{Nat} \to \mathsf{SkipStream}(\mathsf{Nat}) \\
&\langle \mathit{countDown} \| x \cdot \mathsf{Head}[\alpha] \rangle && = \langle \mathsf{Just}(x) \| \alpha \rangle \\
&\langle \mathit{countDown} \| \mathsf{Z} \cdot \mathsf{Tail}[\beta] \rangle && = \langle \mathit{empty} \| \beta \rangle \\
&\langle \mathit{countDown} \| \mathsf{S}(x) \cdot \mathsf{Tail}[\beta] \rangle && = \langle \mathit{countDown} \| x \cdot \beta \rangle \quad \textit{End example } 6.3.
\end{aligned}$$

Example 6.4. As opposed to the co-data approach to describing infinite objects, there is a more widely used approach in lazy functional languages like Haskell and proof assistants like Coq that still favors framing information as data. For example, an infinite list of zeroes is expressed in this functional style by an endless sequence of Cons:

$$\begin{aligned}
\mathit{zeroes} &: \mathsf{List}(\mathsf{Nat}) \\
\mathit{zeroes} &= \mathsf{Cons}\,\mathsf{Z}\,\mathit{zeroes}
\end{aligned}$$

We could emulate this definition in sequent style as the expansion of *zeros* when observed by any α:

$$\begin{aligned}
&\mathit{zeroes} && : \mathsf{List}(\mathsf{Nat}) \\
&\langle \mathit{zeroes} \| \alpha \rangle && = \langle \mathsf{Cons}(\mathsf{Z}, \mathit{zeroes}) \| \alpha \rangle
\end{aligned}$$

Likewise, we can describe the concatenation of two, possibly infinite lists in the same way, by pattern-matching on the call:

$$\begin{aligned}
&\mathit{cat} && : \mathsf{List}(X) \to \mathsf{List}(X) \to \mathsf{List}(X) \\
&\langle \mathit{cat} \| \mathsf{Nil} \cdot ys \cdot \alpha \rangle && = \langle ys \| \alpha \rangle \\
&\langle \mathit{cat} \| \mathsf{Cons}(x, xs) \cdot ys \cdot \alpha \rangle && = \langle \mathsf{Cons}(x, \mu\beta. \langle \mathit{cat} \| xs \cdot ys \cdot \beta \rangle) \| \alpha \rangle
\end{aligned}$$

The intention is that, so long as we do not evaluate the sub-components of Cons eagerly, then α receives a result even if xs is an infinitely long list like *zeroes*.

End example 6.4.

In each of these examples, we were only concerned with writing recursive programs, but have not showed that they always terminate. Termination is especially important for proof assistants and dependently typed languages, which rely on the absence of infinite loops for their logical consistency. If we consider the programs in Examples 6.1 and 6.2, then termination appears fairly straightforward by structural recursion *somewhere* in a function call: each recursive invocation of *length* has a structurally smaller list for the argument, and each recursive invocation of *countUp*, and *countDown* has a smaller stream projection out of its returned result. However, formulating this argument in general turns out to be more complicated. Even worse, the "infinite data structures" in Example 6.4 do not have as clear of a concept of "termination:" *zeroes* and concatenation could go on forever, if they are not given a bound to stop. To tackle these issues, we will phrase principles of well-founded recursion in the parametric $\mu\tilde{\mu}$-calculus, so that we arrive at a core calculus capable of expressing complex termination arguments (parametrically to the chosen evaluation strategy) inside the calculus itself (see Section 6.4).

Polymorphism and Higher Kinds

Before we can talk about statically-guaranteed termination arguments in types, we must first be able to *quantify* over types. That is to say, we need to extend the parametric $\mu\tilde{\mu}$-calculus with type quantifiers like $\forall$ and $\exists$ that we had seen previously in natural deduction (Chapter II) and the sequent calculus (Chapter III). We could just add special connectives with their own separate rules for the quantifiers to the calculus. However, instead let's look at how we can enrich the existing mechanisms of data and co-data to incorporate both $\forall$- and $\exists$-style quantifiers as just more declared (co-)data types like products, sums, and functions.

As it turns out, starting from the multi-kinded parametric sequent calculus from Sections 5.4 and 5.5 we are almost already there. First of all, we will extend the syntax of terms and co-terms to let (co-)data structures contain types in addition to sub-expressions, as shown in Figure 6.1. This change means that the patterns in case abstractions can now bind *type variables* in addition to ordinary (co-)variables, so that (co-)terms can abstract over types as well as other (co-)terms like in the polymorphic λ-calculus (Section 2.2) or polymorphic sequent calculus (Section 3.3). In addition, we will also allow types to abstract over types by extending the language

$$X, Y, Z \in \mathit{TypeVariable} ::= \ldots \qquad \mathcal{R}, \mathcal{S}, \mathcal{T} \in \mathit{BaseKind} ::= \ldots \qquad \mathsf{F}, \mathsf{G} \in \mathit{Connective} ::= \ldots$$
$$k, l \in \mathit{Kind} ::= \mathcal{S} \mid k \to l \qquad A, B, C \in \mathit{Type} ::= X \mid \mathsf{F}(\vec{A}) \mid \lambda X : k.B \mid A\ B$$

$$\begin{aligned}
x, y, z \in \mathit{Variable} &::= \ldots & \alpha, \beta, \gamma \in \mathit{CoVariable} &::= \ldots \\
\mathsf{K} \in \mathit{Constructor} &::= \ldots & \mathsf{O} \in \mathit{Observer} &::= \ldots
\end{aligned}$$
$$\begin{aligned}
c \in \mathit{Command} &::= \langle v \| e \rangle \\
v \in \mathit{Term} &::= x \mid \mu\alpha.c \mid \mathsf{K}^{\vec{A}}(\vec{e}, \vec{v}) \mid \mu\Big(\mathsf{O}^{\overrightarrow{X:k}}[\vec{x}, \vec{\alpha}].c \mid \ldots\Big) \\
e \in \mathit{CoTerm} &::= \alpha \mid \tilde{\mu}x.c \mid \tilde{\mu}\Big[\mathsf{K}^{\overrightarrow{X:k}}(\vec{\alpha}, \vec{x}).c \mid \ldots\Big] \mid \mathsf{O}^{\vec{A}}[\vec{v}, \vec{e}]
\end{aligned}$$

FIGURE 6.1. The syntax of types and programs in the higher-order $\mu\tilde{\mu}$-calculus.

of kinds (denoted by the metavariables k, l) to include *arrow kinds* $k \to l$ in addition to base kinds $\mathcal{S}$, which gives us *type functions* also shown in Figure 6.1. The type-level language of functions uses the notation of the λ-calculus, so that a type function with the parameter $X : k$ is introduced as the λ-abstraction $\lambda X : k.B$ and a type function is applied as $A\ B$.

Intuitively, the motivation for adding type functions to the language is to let (co-)data declarations abstract over them, giving us higher-order (co-)data types. In particular, the addition of type abstraction in both programs and types lets us extend the multi-kinded (co-)data declaration mechanism and kind system as shown in Figure 6.2. The main addition is that now the constructors in a data declaration of $\mathsf{F}(\vec{X})$ and the observers in a co-data declaration of $\mathsf{G}(\vec{X})$can introduce *hidden* quantified type variables $\vec{Y}$ that do not appear in the externally visible interface $\vec{X}$ of the connective. For example, for some fixed kind $\mathcal{S}$, we can give declarations for the universal (∀) and existential (∃) quantification over a type of kind k as follows:

$$\begin{aligned}
&\textbf{codata}\,\forall_k(X : k \to \mathcal{S}) : \mathcal{S}\,\textbf{where} && \textbf{data}\,\exists_k(X : k \to \mathcal{S}) : \mathcal{S}\,\textbf{where} \\
&\quad \_ @ \_ : \Big(\mid \forall_k(X) \vdash^{Y:k} X\ Y : \mathcal{S} \Big) && \quad \_ @ \_ : \Big(X\ Y : \mathcal{S} \vdash^{Y:k} \exists_k(X) \mid \Big)
\end{aligned}$$

These declarations extend the same notion of quantifiers in the dual calculi to higher kinds k, where we use the shorthand $\forall Y{:}k.A$ for $\forall_k(\lambda Y{:}k.A)$ and $\exists Y{:}k.A$ for

[3] As before, this is shorthand for a (co-)data declaration of $\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}$ in $\mathcal{G}$.

$$
\begin{aligned}
\mathit{decl} \in \mathit{Declaration} ::={}& \mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right)} \\
\mid{}& \mathbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(\vec{X}) \vdash^{\overrightarrow{Y:l}} \overrightarrow{B : \mathcal{R}}\right)}
\end{aligned}
$$

$$
\mathcal{G} \in \mathit{GlobalEnv} ::= \overrightarrow{\mathit{decl}} \qquad \Theta \in \mathit{TypeEnv} ::= \overrightarrow{X : k}
$$

$$
\Gamma \in \mathit{InputEnv} ::= \overrightarrow{x : A} \qquad \Delta \in \mathit{OutputEnv} ::= \overrightarrow{\alpha : A}
$$

$$
J, H \in \mathit{Judgement} ::= \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq} \mid (\mathcal{G} \vdash \mathit{decl}) \mid (\Theta \vdash_{\mathcal{G}} A : k)
$$

Declaration rules:

$$
\frac{\overrightarrow{\overrightarrow{X : k}, \overrightarrow{Y : l} \vdash_{\mathcal{G}} A : \mathcal{T}} \quad \overrightarrow{\overrightarrow{X : k}, \overrightarrow{Y : l} \vdash_{\mathcal{G}} B : \mathcal{R}} \quad \overrightarrow{\left(\vdash_{\mathcal{G}}^{\overrightarrow{X:k},\overrightarrow{Y:l}}\right)\mathbf{seq}}}{\mathcal{G} \vdash \begin{array}{c} \mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where} \\ \overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right)} \end{array}} \; \mathit{data}
$$

$$
\frac{\overrightarrow{\overrightarrow{X : k}, \overrightarrow{Y : l} \vdash_{\mathcal{G}} A : \mathcal{T}} \quad \overrightarrow{\overrightarrow{X : k}, \overrightarrow{Y : l} \vdash_{\mathcal{G}} B : \mathcal{R}} \quad \overrightarrow{\left(\vdash_{\mathcal{G}}^{\overrightarrow{X:k},\overrightarrow{Y:l}}\right)\mathbf{seq}}}{\mathcal{G} \vdash \begin{array}{c} \mathbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where} \\ \overrightarrow{\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{F}(\vec{X}) \vdash^{\overrightarrow{Y:l}} \overrightarrow{B : \mathcal{R}}\right)} \end{array}} \; \mathit{codata}
$$

Kind rules:

$$
\frac{}{\Theta, X : k \vdash_{\mathcal{G}} X : k} \; \mathit{TV} \qquad \frac{\overrightarrow{\Theta \vdash_{\mathcal{G}} C : k} \quad (\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S})^3 \in \mathcal{G}}{\Theta \vdash_{\mathcal{G}} \mathsf{F}(\vec{C}) : \mathcal{S}} \; \mathsf{F}T
$$

$$
\frac{\Theta, X : k \vdash_{\mathcal{G}} A : l}{\Theta \vdash_{\mathcal{G}} \lambda X : k.A : k \to l} \; {\to}I^2 \qquad \frac{\Theta \vdash_{\mathcal{G}} A : k \to l \quad \Theta \vdash_{\mathcal{G}} B : k}{\Theta \vdash_{\mathcal{G}} A\ B : l} \; {\to}E^2
$$

Well-formed sequent rules:

$$
\frac{}{(\ \vdash\ )\,\mathbf{seq}} \qquad \frac{\mathcal{G} \vdash \mathit{decl} \quad \left(\ \vdash_{\mathcal{G}}\ \right)\mathbf{seq}}{\left(\ \vdash_{\mathcal{G},\mathit{decl}}\ \right)\mathbf{seq}} \qquad \frac{\left(\ \vdash^{\Theta}_{\mathcal{G}}\ \right)\mathbf{seq}}{\left(\ \vdash^{\Theta,X:k}_{\mathcal{G}}\ \right)\mathbf{seq}}
$$

$$
\frac{\Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}}{\left(\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}} \qquad \frac{\Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}}{\left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)\mathbf{seq}}
$$

FIGURE 6.2. The kind system for the higher-order parametric $\mu\tilde{\mu}$ sequent calculus.

$\exists_k(\lambda Y{:}k.A)$. A term of type $\forall Y{:}k.A$ is introduced as the case abstraction $\mu(Y{:}k \mathbin{@} \alpha.c)$ that is consumed buy the observation $B \mathbin{@} e$. Dually, a term of type $\exists Y{:}\, k.A$ is introduced by the construction $B \mathbin{@} v$ that is consumed by the case abstraction $\tilde{\mu}[Y{:}k \mathbin{@} x.c]$.

Note that the kind system in Figure 6.2 also includes an entirely new kind of judgement $\left(\Gamma \vdash_{\mathcal{G}} \Delta\right) \mathbf{seq}$ that says a general sequent $\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta$ is well-formed. This judgement is now necessary because of the addition of type functions, which are a new kind of type that does not actually classify any term or co-term. In other words, supposing that a free variable x has type $\lambda X{:}\mathcal{S}.X$ would be nonsensical. Therefore, we rule any such possibility by the rules of $\left(\Gamma \vdash_{\mathcal{G}} \Delta\right) \mathbf{seq}$, which enforce that for every $x : A$ in Γ and $\alpha : A$ in Δ, A must belong to some base kind $\mathcal{S}$ and *not* some other kind like $k \to l$. This is the same reason that the declarations for (co-)data types can only declare connectives of the form $\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}$ for some base kind $\mathcal{S}$, and similarly the sequents that give the types of constructors and observers are well-formed whenever the declaration is well-formed according to the *data* and *codata* rules.

Since we have added new forms of terms and co-terms which package up and abstract over types, we also need to update the typing rules to accomodate these new forms in the higher-order parametric $\mu\tilde{\mu}$-calculus, as shown in Figure 6.3. Note that the judgements and core typing rules are exactly the same as the core typing rules for the multi-kinded type system from Figure 5.16 plus the addition of the type conversion rules *TCR* and *TCL*. These conversion rules say that any $\beta =$ equivalent types (in the sense of the typed $\beta\eta$ equational theory of the λ-calculus from Chapter II Section 2.2 and denoted by the judgement $\Theta \vdash_{\mathcal{G}} A =_{\beta\eta} B : \mathcal{S}$ with the rules given in Figure 6.4) contain exactly the same terms and co-terms.

The only other update is in the left and right introduction rules for particular (co-)data types, which now account for the possibility that constructions and observations might include types which are referenced in the components of the pattern. For (co-)data structures, this means that there is a choice of hidden types $\overrightarrow{C_i'}$ which must be substituted for the quantified type variables $\overrightarrow{Y_i : l_i}$ in the sub-(co-)terms of the structure. For (co-)data case abstractions, we need to extend the local type environment Θ with the abstracted type variables, just as we must extend the local input and output environments with the abstracted (co-)variables. For example, the specific instances of the general typing rules for the two families of quantifiers $\forall_k$ and

$$\mathit{Judgement} ::= c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right) \mid (\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta)$$

Type conversion rules:

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta \quad \Theta \vdash_{\mathcal{G}} A =_{\beta\eta} B : \mathcal{S}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : B \mid \Delta}\ \mathit{TCR} \qquad \frac{\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta \quad \Theta \vdash_{\mathcal{G}} A =_{\beta\eta} B : \mathcal{S}}{\Gamma \mid e : B \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathit{TCL}$$

Logical rules:

Given $\mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y_i : l_i}} \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\theta = \left\{\overrightarrow{C/X}\right\} \ \overrightarrow{\Theta \vdash_{\mathcal{G}} C'_i\theta : l_i\theta} \ \ \theta' = \left\{\overrightarrow{C'_i/Y_i}\right\}\theta \ \overrightarrow{\Gamma'_j \mid e : B_{ij}\theta' \vdash^{\Theta}_{\mathcal{G}} \Delta'_j}^{j} \ \ \overrightarrow{\Gamma_j \mid v : A_{ij}\theta' \vdash^{\Theta}_{\mathcal{G}} \Delta_j}^{j}}{\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \vdash^{\Theta}_{\mathcal{G}} \mathsf{K}_i^{\overrightarrow{C'}}(\vec{e}, \vec{v}) : \mathsf{F}(\vec{C}) \mid \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}}\ \mathsf{F}R_{\mathsf{K}_i}$$

$$\frac{\theta = \left\{\overrightarrow{C/X}\right\} \quad \overrightarrow{c_i : \left(\Gamma, \overrightarrow{x_i : A_i\theta} \vdash^{\Theta, \overrightarrow{Y_i : l_i\theta}}_{\mathcal{G}} \overrightarrow{\alpha_i : B_i\theta}, \Delta\right)}^{i}}{\Gamma \mid \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y_i : l_i}}(\vec{\alpha_i}, \vec{x_i}).c_i}^{i}\right] : \mathsf{F}(\vec{C}) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathsf{F}L$$

Given $\mathbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\overrightarrow{X}) \vdash^{\overrightarrow{Y_i : l_i}} \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\theta = \left\{\overrightarrow{C/X}\right\} \quad \overrightarrow{c_i : \left(\Gamma, \overrightarrow{x_i : A_i\theta} \vdash^{\Theta, \overrightarrow{Y_i : l_i\theta}}_{\mathcal{G}} \overrightarrow{\alpha_i : B_i\theta}, \Delta\right)}^{i}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y_i : l_i}}[\vec{x_i}, \vec{\alpha_i}].c_i}^{i}\right) : \mathsf{G}(\vec{C}) \mid \Delta}\ \mathsf{G}R$$

$$\frac{\theta = \left\{\overrightarrow{C/X}\right\} \ \overrightarrow{\Theta \vdash_{\mathcal{G}} C'_i : l_i} \ \ \theta' = \left\{\overrightarrow{C'_i/Y_i}\right\}\theta \ \overrightarrow{\Gamma_j \mid v : A_{ij}\theta' \vdash^{\Theta}_{\mathcal{G}} \Delta_j}^{j} \ \ \overrightarrow{\Gamma'_j \mid e : B_{ij}\theta' \vdash^{\Theta}_{\mathcal{G}} \Delta'_j}^{j}}{\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \mid \mathsf{O}_i^{\overrightarrow{C'_i}}[\vec{v}, \vec{e}] : \mathsf{G}(\vec{C}) \vdash^{\Theta}_{\mathcal{G}} \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}}\ \mathsf{G}L_{\mathsf{O}_i}$$

FIGURE 6.3. Types of higher-order (co-)data in the parametric $\mu\tilde{\mu}$ sequent calculus.

$$\frac{\Theta, X : k \vdash_{\mathcal{G}} A : l \quad \Theta \vdash_{\mathcal{G}} B : k}{\Theta \vdash_{\mathcal{G}} (\lambda X{:}k.A)\ B =_{\beta\eta} A\{B/X\} : l}\ \beta$$

$$\frac{\Theta \vdash_{\mathcal{G}} A : k \to l}{\Theta \vdash_{\mathcal{G}} \lambda X{:}k.A\ X =_{\beta\eta} A : k \to l}\ \eta$$

$$\frac{\Theta \vdash_{\mathcal{G}} A : k}{\Theta \vdash_{\mathcal{G}} A =_{\beta\eta} A : k}\ \mathit{refl} \qquad \frac{\Theta \vdash_{\mathcal{G}} B =_{\beta\eta} A : k}{\Theta \vdash_{\mathcal{G}} A =_{\beta\eta} B : k}\ \mathit{symm}$$

$$\frac{\Theta \vdash_{\mathcal{G}} A =_{\beta\eta} B : k \quad \Theta \vdash_{\mathcal{G}} B =_{\beta\eta} C : k}{\Theta \vdash_{\mathcal{G}} A =_{\beta\eta} C : k}\ \mathit{trans}$$

$$\frac{}{\Theta, X : k \vdash_{\mathcal{G}} X =_{\beta\eta} X : k}\ TV$$

$$\frac{\Theta \vdash_{\mathcal{G}} \mathsf{F}(\overrightarrow{C}) : \mathcal{S} \quad \overrightarrow{\Theta \vdash_{\mathcal{G}} C =_{\beta\eta} C' : k} \quad \Theta \vdash_{\mathcal{G}} \mathsf{F}(\overrightarrow{C'}) : \mathcal{S} \quad (\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}) \in \mathcal{G}}{\Theta \vdash_{\mathcal{G}} \mathsf{F}(\overrightarrow{C}) =_{\beta\eta} \mathsf{F}(\overrightarrow{C'}) : \mathcal{S}}\ \mathsf{F}T$$

$$\frac{\Theta, X : k \vdash_{\mathcal{G}} A =_{\beta\eta} A' : l}{\Theta \vdash_{\mathcal{G}} \lambda X{:}k.A =_{\beta\eta} \lambda X{:}k.A' : k \to l}\ {\to}I^2$$

$$\frac{\Theta \vdash_{\mathcal{G}} A =_{\beta\eta} A' : k \to l \quad \Theta \vdash_{\mathcal{G}} B =_{\beta\eta} B' : k}{\Theta \vdash_{\mathcal{G}} A\ B =_{\beta\eta} A'\ B' : l}\ {\to}E^2$$

FIGURE 6.4. $\beta\eta$ conversion of higher-order types.

$$
\begin{array}{ll}
(\beta^{\mathsf{F}}) & \left\langle \mathsf{K}_i^{\vec{C}}(\vec{e}, \vec{v}) \middle\| \tilde{\mu}\left[\cdots \mid \mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{\alpha}, \vec{x}).c_i \mid \cdots\right] \right\rangle \succ_{\beta^{\mathsf{F}}} \left\langle \mu\vec{\alpha}. \left\langle \vec{v} \middle\| \tilde{\mu}\vec{x}.c_i \left\{\overrightarrow{C/Y}\right\} \right\rangle \middle\| \vec{e} \right\rangle \\
(\beta^{\mathsf{G}}) & \left\langle \mu\left(\cdots \mid \mathsf{O}_i^{\overrightarrow{Y{:}l}}[\vec{x}, \vec{\alpha}].c_i \mid \cdots\right) \middle\| \mathsf{O}_i^{\vec{C}}[\vec{v}, \vec{e}] \right\rangle \succ_{\beta^{\mathsf{G}}} \left\langle \vec{v} \middle\| \tilde{\mu}\vec{x}. \left\langle \mu\vec{\alpha}.c_i \left\{\overrightarrow{C/Y}\right\} \middle\| \vec{e} \right\rangle \right\rangle \\
(\eta^{\mathsf{F}}) & \gamma : \mathsf{F}(\vec{C}) \prec_{\eta^{\mathsf{F}}} \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{\alpha}, \vec{x}).\left\langle \mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{\alpha}, \vec{x}) \middle\| \gamma \right\rangle}^{i}\right] \\
(\eta^{\mathsf{G}}) & z : \mathsf{G}(\vec{C}) \prec_{\eta^{\mathsf{G}}} \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y{:}l}}[\vec{x}, \vec{\alpha}].\left\langle z \middle\| \mathsf{O}_i^{\overrightarrow{Y{:}l}}[\vec{x}, \vec{\alpha}] \right\rangle}^{i}\right)
\end{array}
$$

FIGURE 6.5. The $\beta\eta$ laws for higher-order data and co-data types.

$\exists_k$ above are:

$$
\frac{c : \left(\Gamma \vdash_{\mathcal{G}}^{\Theta, X:k} \alpha : A\ X, \Delta\right)}{\Gamma \vdash_{\mathcal{G}}^{\Theta} \mu(X : k \mathbin{@} \alpha.c) : \forall_k(A) \mid \Delta}\ \forall R_k
\qquad
\frac{\Theta \vdash_{\mathcal{G}} B : k \quad \Gamma \mid e : A\ B \vdash_{\mathcal{G}}^{\Theta} \Delta}{\Gamma \mid B \mathbin{@} e : \forall_k(A) \vdash_{\mathcal{G}}^{\Theta} \Delta}\ \forall L_k
$$

$$
\frac{\Theta \vdash_{\mathcal{G}} B : k \quad \Gamma \vdash_{\mathcal{G}}^{\Theta} v : A\ B \mid \Delta}{\Gamma \vdash_{\mathcal{G}}^{\Theta} B \mathbin{@} v : \exists_k(A) \mid \Delta}\ \exists R_k
\qquad
\frac{c : \left(\Gamma, x : A\ X \vdash_{\mathcal{G}}^{\Theta, X:k} \Delta\right)}{\Gamma \mid \tilde{\mu}[X : k \mathbin{@} x.c] : \exists_k(A) \vdash_{\mathcal{G}}^{\Theta} \Delta}\ \exists L_k
$$

Other than this addition, the rules are the same as before in Section 5.4.

Thus concluding the static semantics of the higher-order parametric $\mu\tilde{\mu}$-calculus, we must also consider how the extension affects the dynamic semantics. The short answer is: not much. In general, the types contained in structures must be substituted for the type variables bound by patterns during pattern-matching, but this does not significantly alter the behavior of a program. More specifically, the core $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}$ theory of substitution does not change at all, since the form of input and output abstractions remain the same, the typed $\beta\eta$ theory of (co-)data accounts for the presence of types in programs as shown in Figure 6.5, where the connectives F and G are declared in $\mathcal{G}$ as in Figure 6.3, and likewise the untyped $\beta\varsigma$ theory of (co-)data is extended as shown in Figure 6.6. We must also extend the inference rules from Figure 5.18 for checking that expressions are well-kinded so that we know which substitution strategy to use when mixing several within a program, as shown in Figure 6.7, by just ignoring the additional type annotations on (co-)data structures. Likewise, the definitions of particular substitution strategies, like $\mathcal{V}$, $\mathcal{N}$, $\mathcal{LV}$, and $\mathcal{LN}$, are only changed by annotating structures and patterns with types and type variables, and otherwise exactly the same as their definitions in Chapter V.

$$
\begin{array}{ll}
(\beta_{\mathcal{S}}) & \left\langle \mathsf{K}^{\vec{C}}(\vec{E},\vec{V}) \middle\| \tilde{\mu}\left[\cdots \mid \mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{\alpha},\vec{x}).c \mid \cdots\right] \right\rangle \succ_{\beta_{\mathcal{S}}} c\left\{\overrightarrow{C/Y}, \overrightarrow{E/\alpha}, \overrightarrow{V/x}\right\} \\
(\beta_{\mathcal{S}}) & \left\langle \mu\left(\cdots \mid \mathsf{O}^{\overrightarrow{Y{:}l}}(\vec{x},\vec{\alpha}).c \mid \cdots\right) \middle\| \mathsf{O}^{\vec{C}}(\vec{E},\vec{V}) \right\rangle \succ_{\beta_{\mathcal{S}}} c\left\{\overrightarrow{C/Y}, \overrightarrow{V/x}, \overrightarrow{E/\alpha}\right\}
\end{array}
$$

$$
\left.
\begin{array}{ll}
(\varsigma_{\mathcal{S}}) & \mathsf{K}^{\vec{C}}(\vec{E},e',\vec{e},\vec{v}) \succ_{\varsigma_{\mathcal{S}}} \mu\alpha.\left\langle \mu\beta.\left\langle \mathsf{K}^{\vec{C}}(\vec{E},\beta,\vec{e},\vec{v}) \middle\| \alpha \right\rangle \middle\| e' \right\rangle \\
(\varsigma_{\mathcal{S}}) & \mathsf{K}^{\vec{C}}(\vec{E},\vec{V},v',\vec{v}) \succ_{\varsigma_{\mathcal{S}}} \mu\alpha.\left\langle v' \middle\| \tilde{\mu}y.\left\langle \mathsf{K}^{\vec{C}}(\vec{E},\vec{V},y,\vec{v}) \middle\| \alpha \right\rangle \right\rangle \\
(\varsigma_{\mathcal{S}}) & \mathsf{O}^{\vec{C}}(\vec{V},v',\vec{v},\vec{e}) \succ_{\varsigma_{\mathcal{S}}} \tilde{\mu}x.\left\langle v' \middle\| \tilde{\mu}y.\left\langle x \middle\| \mathsf{O}^{\vec{C}}(\vec{V},y,\vec{v},\vec{e}) \right\rangle \right\rangle \\
(\varsigma_{\mathcal{S}}) & \mathsf{O}^{\vec{C}}(\vec{V},\vec{E},e',\vec{e}) \succ_{\varsigma_{\mathcal{S}}} \tilde{\mu}x.\left\langle \mu\beta.\left\langle x \middle\| \mathsf{O}^{\vec{C}}(\vec{V},\vec{E},\beta,\vec{e}) \right\rangle \middle\| e' \right\rangle
\end{array}
\right\}
\begin{array}{l}
v' \notin \mathit{Values}_{\mathcal{S}} \\
e' \notin \mathit{CoValues}_{\mathcal{S}} \\
x,y,\alpha,\beta \text{ fresh}
\end{array}
$$

FIGURE 6.6. The parametric $\beta_{\mathcal{S}}\varsigma_{\mathcal{S}}$ laws for arbitrary higher-order data and co-data.

Given $\mathbf{data}\,\mathsf{F}(\overrightarrow{X:k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y{:}l_i}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$, we have:

$$
\frac{\overrightarrow{\Gamma'_j \mid e :: \mathcal{R}_{ij} \vdash_{\mathcal{G}} \Delta'_j}^{j} \quad \overrightarrow{\Gamma_j \mid v :: \mathcal{T}_{ij} \vdash_{\mathcal{G}} \Delta_j}^{j}}{\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \vdash_{\mathcal{G}} \mathsf{K}_i^{\vec{C}}(\vec{e},\vec{v}) :: \mathcal{S} \mid \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}}\ \mathsf{F}R_{\mathsf{K}_i}
\qquad
\frac{\overrightarrow{c_i :: \left(\Gamma, \overrightarrow{x_i :: \mathcal{T}_{ij}} \vdash_{\mathcal{G}} \overrightarrow{\alpha_i :: \mathcal{R}_{ij}}, \Delta\right)}^{i}}{\Gamma \mid \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y{:}l_i}}(\vec{\alpha_i},\vec{x_i}).c_i}^{i}\right] :: \mathcal{S} \vdash_{\mathcal{G}} \Delta}\ \mathsf{F}L
$$

Given $\mathbf{codata}\,\mathsf{G}(\overrightarrow{X:k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\vec{X}) \vdash^{\overrightarrow{Y{:}l_i}} \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$, we have:

$$
\frac{\overrightarrow{c_i :: \left(\Gamma, \overrightarrow{x_i :: \mathcal{T}_{ij}} \vdash_{\mathcal{G}} \overrightarrow{\alpha_i :: \mathcal{R}_{ij}}, \Delta\right)}^{i}}{\Gamma \vdash_{\mathcal{G}} \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y{:}l_i}}[\vec{x_i},\vec{\alpha_i}].c_i}^{i}\right) :: \mathcal{S} \mid \Delta}\ \mathsf{G}R
\qquad
\frac{\overrightarrow{\Gamma_j \mid v :: \mathcal{T}_{ij} \vdash_{\mathcal{G}} \Delta_j}^{j} \quad \overrightarrow{\Gamma'_j \mid e :: \mathcal{R}_{ij} \vdash_{\mathcal{G}} \Delta'_j}^{j}}{\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \mid \mathsf{O}_i^{\vec{C}}[\vec{v},\vec{e}] :: \mathcal{S} \vdash_{\mathcal{G}} \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}}\ \mathsf{G}L_{\mathsf{O}_i}
$$

FIGURE 6.7. Type-agnostic kind system for higher-order multi-kinded (co-)data.

Well-Founded Recursion Principles

There is one fundamental difficulty in ensuring termination for programs written in a sequent calculus style: even incredibly simple programs perform their structural recursion from *within* some larger overall structure. For example, consider the humble *length* function from Example 6.1. The decreasing component in the definition of *length* is clearly the list argument which gets smaller with each call. However, in the sequent calculus, the actual recursive invocation of *length* is the *entire* call-stack. This is because the recursive call to *length* does not return to its original caller, but to some place new. When written in a functional style, this information is implicit since the recursive call to *length* is not a tail-call, but rather $\mathsf{S}(\mathit{length}\ \mathit{xs})$. When written in a sequent style, this extra information becomes an explicit part of the function call structure, necessary to remember to increment the output of the function before ultimately returning. This means that we must carry around enough memory to store our ever increasing result amidst our ever decreasing recursion.

Establishing termination for sequent calculus therefore requires a more finely controlled language for specifying "what's getting smaller" in a recursive program, pointing out *where* the decreasing measure is hidden within recursive invocations. For this purpose, we adopt a type-based approach to termination checking (Abel, 2006). Besides allowing us to abstract over termination-ensuring measures, we can also specify which parts of a complex type are used as part of the termination argument. As a consequence for handling simplistic functions like *length*, we will find that, for free, the calculus ends up as a robust language for describing more advanced recursion over structures, including lexicographic and mutual recursion over both data and co-data structures simultaneously.

In considering the type-based approach to termination in the sequent calculus, we identify two different styles for the type-level measure indices. The first is an exacting notion of index with a predictable structure matching the natural numbers and which we use to perform *primitive recursion*. This style of indexing gives us a tight control over the size of structures and depend on the specific structure of the index in the style of GADTs, allowing us to define types like the fixed-sized vectors of values from dependently typed languages as well as a direct encoding of "infinite" structures as found in lazy functional languages. The second is a looser notion that only tracks the upper bound of indices and which we use to perform *noetherian recursion*. This style of indexing is more in tune with typical structurally recursive programs like *length* and also supports full run-time erasure of bounded indices while still maintaining termination of the index-erased programs.

Primitive Recursion

We begin with the seemingly more basic of the two recursion schemes: primitive recursion on a single natural number index. These natural number indices are used in types in two different ways. First, the indices act as an explicit measure in recursively defined (co-)data types, tracking the recursive sub-components of their structures in the types themselves. Second, the indices are abstracted over by the primitive recursion principle, allowing us to generalize over arbitrary indices and write looping programs. For simplicity, we will limit ourselves to a single arbitrary base kind $\mathcal{S}$ in the discussion to follow, although using multiple different ones is still admissible.

Let's consider some examples of using natural number indices for the purpose of defining (co-)data types with recursive structures. We extend the higher-order (co-)type declaration mechanism from Section 6.2 with the ability to define new (co-)data types by primitive recursion over an index, giving a mechanism for describing recursive (co-)data types with statically tracked measures. Essentially, the constructors are given in two groups—the constructors for the zero case and the constructors for the successor case—and may only contain recursive sub-components at the (strictly) previous index. For example, we may describe vectors of exactly N values of type A, $\mathsf{Vec}(N, A)$, as in dependently typed languages:

$$\begin{array}{lll}
\textbf{data}\,\mathsf{Vec}(i : \mathsf{Ix}, X : \mathcal{S}) : \mathcal{S}\,\textbf{by}\,\text{primitive recursion on } i & & \\
\textbf{where}\,i = 0 & \mathsf{Nil} : & \vdash \mathsf{Vec}(0, X) \mid \\
\textbf{where}\,i = j + 1 & \mathsf{Cons} : & X : \mathcal{S}, \mathsf{Vec}(j, X) : \mathcal{S} \vdash \mathsf{Vec}(j + 1, X) \mid
\end{array}$$

where Ix is the kind of type-level natural number indices. Nil builds an empty vector of type $\mathsf{Vec}(0, A)$, and $\mathsf{Cons}(v, v')$ extends the vector $v' : \mathsf{Vec}(N, A)$ with another element $v : A$, giving us a vector with one more element of type $\mathsf{Vec}(N + 1, A)$. These terms are typed by the right rules for Vec:

$$\frac{}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{Nil} : \mathsf{Vec}(0, A) \mid \Delta}\ \mathsf{Vec}\,R_{\mathsf{Nil}}$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta \quad \Gamma' \vdash^{\Theta}_{\mathcal{G}} v : \mathsf{Vec}(M, A) \mid \Delta'}{\Gamma', \Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{Cons}(v, v') : \mathsf{Vec}(M + 1, A) \mid \Delta', \Delta}\ \mathsf{Vec}\,R_{\mathsf{Cons}}$$

Other than these restrictions on the instantiations of $i : \mathsf{Ix}$ for vectors constructed by Nil and Cons, the typing rules for terms of $\mathsf{Vec}(N, A)$ follow the normal pattern for declared data types.[4] Destructing a vector diverges more from the usual pattern of non-recursive data types. Since the constructors of vector values are put in two separate groups, we have two separate case abstractions to consider, depending on whether the vector is empty or not. On the one hand, to destruct an empty vector, we only have to handle the case for Nil, as given by the co-term $\tilde{\mu}[\mathsf{Nil}.c]$. On the other, destructing a non-empty vector requires us to handle the Cons case, as given by the co-term $\tilde{\mu}[\mathsf{Cons}(x, xs).c]$. These co-terms are typed by the two left rules for Vec—one for both its zero and successor instances:

$$\frac{c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}[\mathsf{Nil}.c] : \mathsf{Vec}(0, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathsf{Vec}\, L_0$$

$$\frac{c : \left(\Gamma, x : A, xs : \mathsf{Vec}(M, A) \vdash^{\Theta}_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}[\mathsf{Cons}(x, xs).c] : \mathsf{Vec}(M + 1, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathsf{Vec}\, L_{+1}$$

As a similar example, we can define a less statically constrained list type by primitive recursion. The IxList indexed data type is just like Vec, except that the Nil constructor is available at both the zero and successor cases:

$$\begin{array}{lrr}
\multicolumn{3}{l}{\mathbf{data}\ \mathsf{IxList}(i : \mathsf{Ix}, X : \mathcal{S})\ \mathbf{by}\ \text{primitive recursion on } i} \\
\mathbf{where}\ i = 0 & \mathsf{Nil} : & \vdash \mathsf{IxList}(0, X) \mid \\
\mathbf{where}\ i = j + 1 & \mathsf{Nil} : & \vdash \mathsf{IxList}(j + 1, X) \mid \\
 & \mathsf{Cons} : & X : \mathcal{S}, \mathsf{IxList}(j, X) : \mathcal{S} \vdash \mathsf{IxList}(j + 1, X) \mid
\end{array}$$

Now, destructing a non-zero $\mathsf{IxList}(N + 1, A)$ requires both cases, as given in the co-term $\tilde{\mu}[\mathsf{Nil}.c \mid \mathsf{Cons}(x, xs).c']$. IxList has three right rules for building terms: for Nil at both 0 and $M + 1$ and for Cons:

$$\frac{}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{Nil} : \mathsf{IxList}(0, A) \mid \Delta}\ \mathsf{IxList}\, R_{\mathsf{Nil}\,0} \qquad \frac{}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{Nil} : \mathsf{IxList}(M + 1, A) \mid \Delta}\ \mathsf{IxList}\, R_{\mathsf{Nil}\,+1}$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta \quad \Gamma' \vdash^{\Theta}_{\mathcal{G}} v : \mathsf{IxList}(M, A) \mid \Delta'}{\Gamma', \Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{Cons}(v, v') : \mathsf{IxList}(M + 1, A) \mid \Delta', \Delta}\ \mathsf{IxList}\, R_{\mathsf{Cons}}$$

[4] We can have a vector with an abstract index if we don't yet know what shape it has, as with the variable x or abstraction $\mu\alpha.c$ of type $\mathsf{Vec}(i, A)$.

It also has two left rules: one for case abstractions handling the constructors of the 0 case and another for the $M + 1$ case:

$$\frac{c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}[\mathsf{Nil}.c] : \mathsf{IxList}(0, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathsf{IxList}\ L_0$$

$$\frac{c_0 : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right) \quad c_1 : \left(\Gamma, x : A, xs : \mathsf{Vec}(M, A) \vdash^{\Theta}_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}[\mathsf{Nil}.c_0 \mathsf{Cons}(x, xs).c_1] : \mathsf{IxList}(M + 1, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathsf{IxList}\ L_{+1}$$

To write looping programs over these indexed recursive types, we use a recursion scheme which abstracts over the index occurring anywhere within an arbitrary type. As the types themselves are defined by primitive recursion over a natural number, the recursive structure of programs will also follow the same pattern. The trick then is to embody the primitive induction principle for proving a proposition P over natural numbers:

$$P[0] \wedge (\forall j : \mathbb{N}.P[j] \to P[j + 1]) \to (\forall i : \mathbb{N}.P[i])$$

and likewise the refutation of such a statement, as is given by any specific counter-example—$n : \mathbb{N} \wedge \overline{P[n]} \to \overline{(\forall i : \mathbb{N}.P[i])}$—into logical rules of the sequent calculus.[5] Recall from the reading of sequents in Chapter III, proofs come to the right of entailment ($\vdash A$ means "A is true"), whereas refutations come to the left ($A \vdash$ means "A is false"). Because we will have several recursion principles, we denote this particular one as $\forall$ quantification over Ix, $\forall_{\mathsf{Ix}}$, so that the primitive recursive proposition $\forall i : \mathbb{N}.P[i]$ on natural numbers corresponds to the type $\forall i : \mathsf{Ix}.A$ which is shorthand for $\forall_{\mathsf{Ix}}(\lambda i : \mathsf{Ix}.A)$ with the following inference rules:

$$\frac{\vdash A\ 0 \quad A\ j \vdash_{j:\mathsf{Ix}} A\ (j + 1)}{\vdash \forall_{\mathsf{Ix}}(A)} \qquad \frac{\vdash M : \mathsf{Ix} \qquad A\ M \vdash}{\forall_{\mathsf{Ix}}(A) \vdash}$$

We use this translation of primitive induction into logical rules as the basis for our primitive recursive *co-data type*. The refutation of primitive recursion is given as a specific *counter-example*, so the co-term is a specific construction. Whereas, proof by primitive recursion is a *process* given by *cases*, the term performs case analysis over its observations. The canonical counter-example is described by the co-data type

[5] We use the overbar notation, $\overline{P}$, to denote that the proposition P is false. The use of this notation is to emphasize that we are not talking about negation as a logical connective, but rather the *dual* to a proof that P is true, which is a refutation of P demonstrating that it is false.

declaration for $\forall_{\mathsf{Ix}}$:

$$\begin{aligned} &\mathbf{codata}\, \forall_{\mathsf{Ix}}(X : \mathsf{Ix} \to \mathcal{S}) : \mathcal{S}\, \mathbf{where} \\ &\quad \_ @ \_ : \Big(\mid \forall_{\mathsf{Ix}}(X) \vdash^{j:\mathsf{Ix}} X\ j : \mathcal{S} \Big) \end{aligned}$$

Notice that this is exactly the same co-data definition of $\forall$ quantification from Section 6.2 except that the generic kind k has been specialized to Ix. Therefore, the general mechanism for co-data automatically generates the same left rule for constructing the counter-example, and a right rule for extracting the parts of this construction. However, to give a recursive process for $\forall_{\mathsf{Ix}}$, we need an additional right rule that gives us access to the recursive argument by performing case analysis on the particular index. This scheme for primitive recursion is expressed by the term $\mu(0{:}\mathsf{Ix} @ \alpha.c_0 \mid j+1{:}\mathsf{Ix} @_x \alpha.c_1)$ which performs case analysis on type-level indices at *run-time*, and which can access the recursive result through the extra variable x in the successor pattern $j+1{:}\mathsf{Ix} @_x \alpha$. This term has the typing rule:

$$\frac{c_0 : \Big(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A\ 0, \Delta\Big) \quad c_1 : \Big(\Gamma, x : A\ j \vdash^{\Theta, j:\mathsf{ix}}_{\mathcal{G}} \alpha : A\ (j+1), \Delta\Big)}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu(0{:}\mathsf{Ix} @ \alpha.c_0 \mid j+1{:}\mathsf{Ix} @_x \alpha.c_1) : \forall_{\mathsf{Ix}}(A) \mid \Delta}\ \forall_{\mathsf{Ix}} Rrec$$

Note that this extension of the $\forall_{\mathsf{Ix}}$ connective is allowed by the pragmatist view of co-data types: the observations of a co-data type are fixed up front, but the terms can be "whatever works" with respect to those observations. Terms of type $\forall i{:}\mathsf{Ix}.A$ describe a process which is able to produce $A\,\{N/i\}$, for any index N, by stepwise producing $A\,\{0/i\}$, $A\,\{1/i\}$, $\dots$, $A\,\{N/i\}$ and piping the previous output to the recursive input x of the next step, thus "inflating" the index in the result arbitrarily high. In essence, this follows the interface of an infinitary & (an additive conjunction) of the form $A\,\{0/i\}\ \&\ A\,\{1/i\}\ \&\ A\,\{2/i\}\ \&\ \dots$. The index of the particular step being handled is part of the observer pattern, so that the recursive case abstraction knows which branch to take. In contrast, co-terms of type $\forall i{:}\mathsf{Ix}.A$ *hide* the particular index at which they can consume an input, thereby forcing their input to work for any index.

By just applying duality in the sequent calculus and flipping everything about the turnstyles, we get the opposite notion of primitive recursion as a data type. In particular, we get the data declaration describing a dual type, named $\exists_{\mathsf{Ix}}$:

$$\begin{aligned} &\mathbf{data}\, \exists_{\mathsf{Ix}}(X : \mathsf{Ix} \to \mathcal{S})\, \mathbf{where} \\ &\quad \_ @ \_ : \Big(X\ j : \mathcal{S} \vdash^{j:\mathsf{Ix}} \exists_{\mathsf{Ix}}(X) \mid \Big) \end{aligned}$$

Again, note that this data declaration is just the Ix instance of the general $\exists$ quantifier from Section 6.2. The general mechanism for data automatically generates the right rule for constructing an index-witnessed example case, and a left rule for extracting the index and value from this structure. Further, as before we need an additional left rule for performing self-referential recursion for consuming such a construction:

$$\frac{c_0 : \left(\Gamma, x : A\ 0 \vdash^{\Theta}_{\mathcal{G}} \Delta\right) \quad c_1 : \left(\Gamma, x : A\ (j+1) \vdash^{\Theta, j:\mathsf{Ix}}_{\mathcal{G}} \alpha : A\ j, \Delta\right)}{\Gamma \mid \tilde{\mu}[0{:}\mathsf{Ix} \mathbin{@} x.c_0 \mid j+1{:}\mathsf{Ix} \mathbin{@_{\alpha}} x.c_1] : \exists_{\mathsf{Ix}}(A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \exists_{\mathsf{Ix}} L_{rec}$$

This extension of the $\exists_{\mathsf{Ix}}$ connective is allowed by the verificationist view of data types: the constructions of a data type are fixed up front, but the co-terms can be "whatever works" with respect to those constructions. Dual to before, the recursive output sink can be accessed through the extra co-variable α in the pattern $j + 1{:}\mathsf{Ix} \mathbin{@_{\alpha}} x$. The terms of type $\exists_{\mathsf{Ix}} i{:}\mathsf{Ix}.A$ hide the particular index at which they produce an output. In contrast, it is now the co-terms of the type $\exists_{\mathsf{Ix}} i{:}\mathsf{Ix}.A$ which describe a process which is able to consume $A\,\{N/i\}$ for any choice of N in steps by consuming $A\,\{N/i\}$, ..., $A\,\{0/i\}$ and piping the previous input to the recursive output α of the next step, thus "deflating" the index in the input down to 0. In essence, this follows the interface of an infinitary $\oplus$ (an additive disjunction) of the form $A\,\{0/i\} \oplus A\,\{1/i\} \oplus A\,\{2/i\} \oplus \dots$.

Noetherian Recursion

We now consider the more complex of the two recursion schemes: noetherian recursion over well-ordered indices. As opposed to ensuring a decreasing measure by matching on the specific structure of the index, we will instead quantify over arbitrary indices that are less than the current one. In other words, the details of what these indices look like are not important. Instead, they are used as arbitrary upper bounds in an ever decreasing chain, which stops when we run out of possible indices below our current one as guaranteed by the well-foundedness of their ordering. Intuitively, we may jump by leaps and bounds down the chain, until we run out of places to move. Qualitatively, this different approach to recursion measures allows us to abstract parametrically over the index, and generalize so strongly over the difference in the steps to the point where the particular chosen index is unknown. Thus, because a process receiving a bounded index has so little knowledge of what it looks like, the index cannot influence its action, thereby allowing us to totally erase bounded indices during run-time.

Now let's see how to define some types by noetherian recursion on an ordered index. Unlike primitive recursion, we do not need to consider the possible cases for the chosen index. Instead, we quantify over any index which is *less* than the given one. For example, recall the recursive definition of the Nat data type from Example 6.1. We can be more explicit about tracking the recursive sub-structure of the constructors by indexing Nat with some ordered type, and ensuring that each recursive instance of Nat has a *smaller* index, so that we may define natural numbers by noetherian recursion over ordered indices from a new kind called Ord:

$$\begin{aligned}
&\textbf{data}\,\mathsf{Nat}(i:\mathsf{Ord})\,\textbf{by}\,\text{noetherian recursion on } i\,\textbf{where}\\
&\quad \mathsf{Z}: \qquad\qquad \vdash \mathsf{Nat}(i) \mid\\
&\quad \mathsf{S}: \;\; \mathsf{Nat}(j) \vdash^{j<i} \mathsf{Nat}(i) \mid
\end{aligned}$$

Note that the kind of indices less than i is denoted by $(< i)$, and we write $j < i$ as shorthand for $j : (< i)$. Noetherian recursion in types is surprisingly more straightforward than primitive recursion, and more closely follows the established pattern for data type declarations:

$$\frac{}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{Z} : \mathsf{Nat}(N) \mid \Delta}\ \mathsf{Nat}R_{\mathsf{Z}}$$

$$\frac{\Theta \vdash_{\mathcal{G}} M : < N \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v : \mathsf{Nat}(M) \mid \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{S}^{M}(v) : \mathsf{Nat}(N) \mid \Delta}\ \mathsf{Nat}R_{\mathsf{S}}$$

Z builds a $\mathsf{Nat}(N)$ for any Ord index N, and $\mathsf{S}^{M}(v)$ builds an incremented $\mathsf{Nat}(N)$ out of a $\mathsf{Nat}(M)$, when $M < N$. To destruct a $\mathsf{Nat}(N)$, for any index N, we have the one case abstraction that handles both the Z and S cases:

$$\frac{c_0 : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right) \quad c_1 : \left(\Gamma, x : \mathsf{Nat}(j) \vdash^{\Theta, j<N}_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}\left[\mathsf{Z}.c_0 \mid \mathsf{S}^{j<N}(x).c_1\right] : \mathsf{Nat}(N) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathsf{Nat}L$$

Like the case abstraction for tearing down an existentially constructed value, the pattern for S introduces the free type variable j which stands for an arbitrary index less than N.

We can consider some other examples of (co-)data types defined by noetherian recursion. The definition of finite lists is just an annotated version of the definition from Example 6.1:

$$
\begin{array}{rl}
\multicolumn{2}{c}{\textbf{data}\,\mathsf{List}(i : \mathsf{Ord}, X : \mathcal{S}) : \mathcal{S}\,\textbf{by}\,\text{noetherian recursion on } i\,\textbf{where}} \\
\mathsf{Nil} : & \qquad\qquad\qquad\qquad\quad \vdash \mathsf{List}(i, X) \mid \\
\mathsf{Cons} : & X : \mathcal{S}, \mathsf{List}(j, X) : \mathcal{S} \vdash^{j<i} \mathsf{List}(i, X) \mid
\end{array}
$$

Furthermore, the infinite streams from Example 6.2 can also be defined as a co-data type by noetherian recursion:

$$
\begin{array}{rl}
\multicolumn{2}{c}{\textbf{codata}\,\mathsf{Stream}(i : \mathsf{Ord}, X : \mathcal{S}) : \mathcal{S}\,\textbf{by}\,\text{noetherian recursion on } i\,\textbf{where}} \\
\mathsf{Head} : & \mid \mathsf{Stream}(i, X) \vdash X : \mathcal{S} \\
\mathsf{Tail} : & \mid \mathsf{Stream}(i, X) \vdash^{j<i} \mathsf{Stream}(j, a) : \mathcal{S}
\end{array}
$$

Recursive co-data types follow the dual pattern as data types, with finitely built observations and values given by case analysis on their observations. For $\mathsf{Stream}(N, A)$, we can always ask for the Head of the stream if we have some use for an input of type A, and we can ask for its tail if we can use an input of type $\mathsf{Stream}(M, A)$, for some smaller index $M < N$:

$$
\frac{\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta}{\Gamma \mid \mathsf{Head}[e] : \mathsf{Stream}(N, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\,\mathsf{Stream}L_{\mathsf{Head}}
$$

$$
\frac{\Theta \vdash_{\mathcal{G}} M : < N \quad \Gamma \mid e : \mathsf{Stream}(M, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}{\Gamma \mid \mathsf{Tail}^{M}[e] : \mathsf{Stream}(N, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\,\mathsf{Stream}L_{\mathsf{Tail}}
$$

Whereas a $\mathsf{Stream}(N, A)$ value is given by pattern-matching on these two possible observations:

$$
\frac{c_0 : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right) \quad c_1 : \left(\Gamma \vdash^{\Theta, j<N}_{\mathcal{G}} \beta : \mathsf{Stream}(j, A), \Delta\right)}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\left(\mathsf{Head}[\alpha].c_0 \mid \mathsf{Tail}^{j<N}[\beta].c_1\right) : \mathsf{Stream}(N, A) \mid \Delta}\,\mathsf{Stream}R
$$

As before, to write looping programs over recursive types with bounded indices, we use an appropriate recursion scheme for abstracting over the type index. The proof principle for noetherian induction by a well-founded relation $<$ on a set of ordinals $\mathbb{O}$

is:

$$(\forall j : \mathbb{O}.(\forall i < j.P[i]) \to P[j]) \to (\forall i : \mathbb{O}.P[i])$$

which can be made more uniform by introducing an upper-bound to the quantifier in the conclusion as well as in the hypothesis:

$$(\forall j < n.(\forall i < j.P[i]) \to P[j]) \to (\forall i < n. \to P[i])$$

Likewise, a disproof of this argument is again a witness of a counter-example within the chosen bound:

$$m < n \wedge \overline{P[n]} \to \overline{(\forall i < n.P[i])}$$

We can then translate these principles into inference rules in the sequent calculus, where we represent this new recursion scheme by a co-data type $\forall_{<\mathsf{Ord}}$:

$$\frac{\forall_{<\mathsf{Ord}}(j, A) \vdash_{j<N} A\ j}{\vdash \forall_{<\mathsf{Ord}}(N, A)} \qquad\qquad \frac{\vdash M < N \quad A\ M \vdash}{\forall_{<\mathsf{Ord}}(N, A) \vdash}$$

We will write $\forall i < N.A$ as shorthand for the type $\forall_{<\mathsf{Ord}}(N, \lambda i : \mathsf{Ord}.A)$. We use a similar reading of these rules as a basis for noetherian recursion as we did for primitive recursion. A refutation is still a specific counter-example, so it is represented as a constructed co-term, whereas a proof is a process so is given as a term defined by matching on its observation. Thus, we declare $\forall_{<\mathsf{Ord}}$ as a co-data type of the form:

$$\begin{gathered}\textbf{codata}\,\forall_{<\mathsf{Ord}}(i : \mathsf{Ord}, X : \mathsf{Ord} \to \mathcal{S}) : \mathcal{S}\,\textbf{where}\\ \_\,@\,\_ : \left(\mid \forall_{<\mathsf{Ord}}(i, X) \vdash^{j<i} X\ j : \mathcal{S}\right)\end{gathered}$$

Note that, while very similar, this is slightly different than just the Ord instance of the general $\forall$ quantifier of a generic kind k, because the $\forall_{<\mathsf{Ord}}$ connective also accepts an additional type parameter $i : \mathsf{Ord}$ which serves as the *upper bound* of the quantification. Again, the general mechanism for co-data types tells us how to construct the counter-example with the observation $M \mathbin{@} e$, and destruct it by simple case analysis. The recursive form of case analysis is given manually as the term $\mu(j < N \mathbin{@_x} \alpha.c)$, where x in the pattern is a self-referential variable standing in for the term itself. The typing rule for this recursive case analysis restricts access to itself by making the type of the self-referential variable have a smaller upper bound:

$$\frac{\Theta \vdash_{\mathcal{G}} M :< N \quad \Gamma \mid e : A\ M \vdash^{\Theta}_{\mathcal{G}} \Delta}{\Gamma \mid M \mathbin{@} e : \forall_{<\mathsf{Ord}}(N, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \forall_{<\mathsf{Ord}} L$$

$$\frac{c : \left(\Gamma, x : \forall_{<\mathsf{Ord}}(j, A) \vdash^{\Theta, j<N}_{\mathcal{G}} \alpha : A\ j, \Delta\right)}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu(j < N \mathbin{@}_x \alpha.c) : \forall_{<\mathsf{Ord}}(N, A) \mid \Delta}\ \forall_{<\mathsf{Ord}} R_{rec}$$

In essence, the terms of type $\forall i < N.A$ describe a process which is capable of producing $A\left\{M/i\right\}$ for any $M < N$ by leaps and bounds: an output of type $A\left\{M/i\right\}$ is built up by repeating the same process whenever it is necessary to ascending to an index under M. In contrast, and similar to primitive recursion, co-terms of type $\forall i < N.A$ hide the chosen index, forcing their input to work for any index.

As always, the symmetry of sequents points us to the dual formulation of noetherian recursion in programs. Specifically, we get the dual data type, named $\exists_{<\mathsf{Ord}}$, with the usual shorthand $\exists j < N.A$ for $\exists_{<\mathsf{Ord}}(N, \lambda i{:}\mathsf{Ord}.A)$, via the following data declaration:

$$\begin{gathered}\mathbf{data}\, \exists_{<\mathsf{Ord}}(i : \mathsf{Ord}, X : \mathsf{Ord} \to \mathcal{S}) : \mathcal{S}\, \mathbf{where} \\ \_ \mathbin{@} \_ : \left(X\ j : \mathcal{S} \vdash^{j<i} \exists_{<\mathsf{Ord}}(i, X) \mid \right)\end{gathered}$$

$$\frac{A\left\{j/i\right\} \vdash_{j<N} \exists_{<\mathsf{Ord}} i < j.A}{\exists_{<\mathsf{Ord}} i < N.A \vdash} \qquad \frac{\vdash A\left\{M/i\right\} \qquad \vdash M < N}{\vdash \exists_{<\mathsf{Ord}} i < N.A}$$

Also dual to the $\forall_{<\mathsf{Ord}}$ connective, we have the following pattern construction of $\exists_{<\mathsf{Ord}}$ values as well as the recursive form of pattern-matching where the α in the pattern $j < N \mathbin{@}_\alpha x$ refers to the case abstraction itself:

$$\frac{\Theta \vdash_{\mathcal{G}} M :< N \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v : A\ M \mid \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{G}} M \mathbin{@} v : \exists_{<\mathsf{Ord}}(N, A) \mid \Delta}\ \exists_{<\mathsf{Ord}} R$$

$$\frac{c : \left(\Gamma, x : A\ j \vdash^{\Theta, j<N}_{\mathcal{G}} \alpha : \exists_{<\mathsf{Ord}}(j, A), \Delta\right)}{\Gamma \mid \tilde{\mu}[j < N \mathbin{@}_\alpha x.c] : \exists_{<\mathsf{Ord}}(N, A) \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \exists_{<\mathsf{Ord}} L_{rec}$$

Now that the roles are reversed, the terms of $\exists i < N.A$ hide the chosen index M at which they can produce a result of type $A\left\{M/i\right\}$. Instead, the co-terms of $\exists i < N.A$ consume an $A\left\{M/i\right\}$ for any index $M < N$: an input of type $A\left\{M/i\right\}$ is broken down by repeating the same process whenever it is necessary to descend from an index under M.

Indexed Recursion in the Sequent Calculus

We now flesh out the rest of the system for recursive types and structures for representing recursive programs in the higher-order parametric $\mu\tilde{\mu}$ sequent calculus. The extended syntax for programs, types, and kinds is shown in Figure 6.8, which extends the basic higher-order syntax from Figure 6.1 with size kinds (Ix, Ord, and $< M$), size types (0, $M+1$, and ∞), recursive forms of (co-)data declarations (both primitive and noetherian), and the special recursive forms of case abstraction for the primitive ($\forall_{\mathsf{Ix}}$ and $\exists_{\mathsf{Ix}}$), and noetherian ($\forall_{<\mathsf{Ord}}$ and $\exists_{<\mathsf{Ord}}$) recursion principles. The rules for sorting, kinding, and well-formed sequents, are given in Figure 6.9. Note that the rules for the inequality of Ord, $M < N$, are enough to derive expected facts like $\vdash 4 < 6$, but not so strong that they force us to consider Ord types above ∞. Specifically, the requirement that every Ord has a larger successor, $M < M+1$, *only* when there is an upper bound already established, $M < N$, prevents us from introducing $\infty < \infty + 1$. Additionally, we have two sorts of kinds, those of *erasable* types, $\square$, and *non-erasable* types, $\blacksquare$. Types (of some base kind $\mathcal{S}$) for program-level (co-)values and Ord indices are erasable, because they cannot influence the behavior of a program, whereas the Ix indices are used to drive primitive recursion, and cannot be erased. The fact that there are two sorts of kinds means that some kinds can be ill-sorted (analogous to the possibility of ill-typed programs), so we must add an additional premise to the $\to E^2$ rule checking that the arrow kind $k \to l$ is well-sorted, as well as to the corresponding type conversion rules for function application from Figure 6.4:

$$\frac{\Theta, X : k \vdash_{\mathcal{G}} A : l \quad \Theta \vdash_{\mathcal{G}} B : k \quad \Theta \vdash_{\mathcal{G}} k \to l : \square}{\Theta \vdash_{\mathcal{G}} (\lambda X{:}k.A)\ B =_{\beta\eta} A\{B/X\} : l}\ \beta$$

$$\frac{\Theta \vdash_{\mathcal{G}} A =_{\beta\eta} A' : k \to l \quad \Theta \vdash_{\mathcal{G}} B =_{\beta\eta} B' : k \quad \Theta \vdash_{\mathcal{G}} k \to l : \square}{\Theta \vdash_{\mathcal{G}} A\ B =_{\beta\eta} A'\ B' : l}\ \to E^2$$

Thus, this sorting system separates erasable and non-erasable type annotations found in programs.

$$\begin{aligned}
\mathcal{R}, \mathcal{S}, \mathcal{T} &\in \mathit{BaseKind} ::= \dots \\
X, Y, Z, i, j &\in \mathit{TypeVariable} ::= \dots \qquad \mathsf{F}, \mathsf{G} \in \mathit{Connective} ::= \dots \\
s \in \mathit{Sort} ::= \square \mid \blacksquare \qquad k, l &\in \mathit{Kind} ::= \mathcal{S} \mid k \to l \mid \mathsf{Ix} \mid \mathsf{Ord} \mid (< M) \\
A, B, C, M, N &\in \mathit{Type} ::= X \mid \mathsf{F}(\vec{A}) \mid \lambda X : k.B \mid A\ B \mid 0 \mid M + 1 \mid \infty
\end{aligned}$$

$$\begin{aligned}
\mathit{decl} \in \mathit{Declaration} ::={}& \textbf{data}\, \mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\, \textbf{where}\, \overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right)} \\
&\mid \textbf{codata}\, \mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\, \textbf{where}\, \overrightarrow{\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(\vec{X}) \vdash^{\overrightarrow{Y:l}} \overrightarrow{B : \mathcal{R}}\right)} \\
&\mid \textbf{data}\, \mathsf{F}(i : \mathsf{Ix}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{by}\, \text{primitive recursion on } i \\
&\quad \textbf{where}\, i = 0 \qquad\qquad \overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(0, \vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right)} \\
&\quad \textbf{where}\, i = j + 1 \quad \overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(j + 1, \vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right)} \\
&\mid \textbf{codata}\, \mathsf{G}(i : \mathsf{Ix}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{by}\, \text{primitive recursion on } i \\
&\quad \textbf{where}\, i = 0 \qquad\qquad \overrightarrow{\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(0, \vec{X}) \vdash^{\overrightarrow{Y:l}} \overrightarrow{B : \mathcal{R}}\right)} \\
&\quad \textbf{where}\, i = j + 1 \quad \overrightarrow{\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(j + 1, \vec{X}) \vdash^{\overrightarrow{Y:l}} \overrightarrow{B : \mathcal{R}}\right)} \\
&\mid \textbf{data}\, \mathsf{F}(i : \mathsf{Ord}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{by}\, \text{noetherian recursion on } i \\
&\quad \textbf{where} \qquad\qquad \mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(i, \vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right) \\
&\mid \textbf{codata}\, \mathsf{G}(i : \mathsf{Ord}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{by}\, \text{noetherian recursion on } i \\
&\quad \textbf{where} \qquad\qquad \mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(i, \vec{X}) \vdash^{\overrightarrow{Y:l}} \overrightarrow{B : \mathcal{R}}\right)
\end{aligned}$$

$$\begin{aligned}
x, y, z &\in \mathit{Variable} ::= \dots \qquad \alpha, \beta, \gamma \in \mathit{CoVariable} ::= \dots \\
\mathsf{K} &\in \mathit{Constructor} ::= \dots \qquad\quad \mathsf{O} \in \mathit{Observer} ::= \dots \\
c &\in \mathit{Command} ::= \langle v \| e \rangle \\
v &\in \mathit{Term} ::= x \mid \mu\alpha.c \mid \mathsf{K}^{\vec{A}}(\vec{e}, \vec{v}) \mid \mu\Big(\mathsf{O}^{\overrightarrow{X:k}}[\vec{x}, \vec{\alpha}].c \mid \dots\Big) \\
&\qquad \mid \mu(j{<}M \mathbin{@_x} \alpha.c) \mid \mu(0{:}\mathsf{Ix} \mathbin{@} \alpha.c_0 \mid i{+}1{:}\mathsf{Ix} \mathbin{@_x} \alpha.c_1) \\
e &\in \mathit{CoTerm} ::= \alpha \mid \tilde{\mu}x.c \mid \tilde{\mu}\Big[\mathsf{K}^{\overrightarrow{X:k}}(\vec{\alpha}, \vec{x}).c \mid \dots\Big] \mid \mathsf{O}^{\vec{A}}[\vec{v}, \vec{e}] \\
&\qquad \mid \tilde{\mu}[j{<}M \mathbin{@_\alpha} x.c] \mid \tilde{\mu}[0{:}\mathsf{Ix} \mathbin{@} x.c_0 \mid i + 1{:}\mathsf{Ix} \mathbin{@_\alpha} x.c_1]
\end{aligned}$$

FIGURE 6.8. The syntax of recursion in the higher-order $\mu\tilde{\mu}$-calculus.

$$\mathcal{G} \in \mathit{GlobalEnv} ::= \overrightarrow{decl} \qquad\qquad \Theta \in \mathit{TypeEnv} ::= \overrightarrow{X : k}$$

$$\Gamma \in \mathit{InputEnv} ::= \overrightarrow{x : A} \qquad\qquad \Delta \in \mathit{OutputEnv} ::= \overrightarrow{\alpha : A}$$

$$J, H \in \mathit{Judgement} ::= \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq} \mid (\Theta \vdash_{\mathcal{G}} k : s) \mid (\Theta \vdash_{\mathcal{G}} A : k)$$

Sort rules:

$$\frac{}{\Theta \vdash_{\mathcal{G}} \mathcal{S} : \square} \qquad \frac{}{\Theta \vdash_{\mathcal{G}} \mathsf{Ix} : \blacksquare} \qquad \frac{}{\Theta \vdash_{\mathcal{G}} \mathsf{Ord} : \square}$$

$$\frac{\Theta \vdash_{\mathcal{G}} k : \blacksquare \quad \Theta \vdash_{\mathcal{G}} l : \square}{\Theta \vdash_{\mathcal{G}} k \to l : \square} \qquad \frac{\Theta \vdash_{\mathcal{G}} M : \mathsf{Ord}}{\Theta \vdash_{\mathcal{G}} < M : \square} \qquad \frac{\Theta \vdash_{\mathcal{G}} k : \square}{\Theta \vdash_{\mathcal{G}} k : \blacksquare}$$

Kind rules:

$$\frac{(X : k) \notin \Theta'}{\Theta, X : k, \Theta' \vdash_{\mathcal{G}} X : k}\ TV \qquad \frac{\overrightarrow{\Theta \vdash_{\mathcal{G}} C : k} \quad (\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}) \in \mathcal{G}}{\Theta \vdash_{\mathcal{G}} \mathsf{F}(\overrightarrow{C}) : \mathcal{S}}\ \mathsf{F}T$$

$$\frac{\Theta, X : k \vdash_{\mathcal{G}} A : l}{\Theta \vdash_{\mathcal{G}} \lambda X : k.A : k \to l}\ {\to}I^2 \qquad \frac{\Theta \vdash_{\mathcal{G}} A : k \to l \quad \Theta \vdash_{\mathcal{G}} B : k \quad \Theta \vdash_{\mathcal{G}} k \to l : \square}{\Theta \vdash_{\mathcal{G}} A\ B : l}\ {\to}E^2$$

$$\frac{}{\Theta \vdash_{\mathcal{G}} 0 : \mathsf{Ix}} \qquad \frac{\Theta \vdash_{\mathcal{G}} M : \mathsf{Ix}}{\Theta \vdash_{\mathcal{G}} M + 1 : \mathsf{Ix}} \qquad \frac{}{\Theta \vdash_{\mathcal{G}} \infty : \mathsf{Ord}} \qquad \frac{}{\Theta \vdash_{\mathcal{G}} 0 : < \infty}$$

$$\frac{\Theta \vdash_{\mathcal{G}} M : < \infty}{\Theta \vdash_{\mathcal{G}} M + 1 : < \infty} \qquad \frac{\Theta \vdash_{\mathcal{G}} N : \mathsf{Ord} \quad \Theta \vdash_{\mathcal{G}} M : < N}{\Theta \vdash_{\mathcal{G}} M : < M + 1}$$

$$\frac{\Theta \vdash_{\mathcal{G}} M : < M' \quad \Theta \vdash_{\mathcal{G}} M' : < N}{\Theta \vdash_{\mathcal{G}} M : < N} \qquad \frac{\Theta \vdash_{\mathcal{G}} N : \mathsf{Ord} \quad \Theta \vdash_{\mathcal{G}} M : < N}{\Theta \vdash_{\mathcal{G}} M : \mathsf{Ord}}$$

Well-formed sequent rules:

$$\frac{}{(\ \vdash\ )\,\mathbf{seq}} \qquad \frac{\mathcal{G} \vdash decl \quad \left(\ \vdash_{\mathcal{G}}\ \right)\mathbf{seq}}{\left(\ \vdash_{\mathcal{G}, decl}\ \right)\mathbf{seq}} \qquad \frac{\Theta \vdash_{\mathcal{G}} k : s \quad \left(\ \vdash^{\Theta}_{\mathcal{G}}\ \right)\mathbf{seq}}{\left(\ \vdash^{\Theta, X:k}_{\mathcal{G}}\ \right)\mathbf{seq}}$$

$$\frac{\Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}}{\left(\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}} \qquad \frac{\Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}}{\left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)\mathbf{seq}}$$

FIGURE 6.9. The kind system for size-indexed higher-order $\mu\tilde{\mu}$ sequent calculus.

Before admitting a user-defined (co-)data type into the system, we need to check that its declaration actually denotes a meaningful type via the judgement $\mathcal{G} \vdash \mathit{decl}$, so we must extend Figure 6.2 with additional rules for when recursively-defined (co-)data types are well-formed. When checking for well-formedness of (co-)data types defined by primitive induction on $i : \mathsf{Ix}$, as with the general form

$$\begin{array}{lll}
\textbf{data}\, \mathsf{F}(i : \mathsf{Ix}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{by}\, \text{primitive recursion on } i & & \\
\textbf{where}\, i = 0 & \mathsf{K}_1 : \quad \overrightarrow{B_1 : \mathcal{T}_1} \vdash_{\overrightarrow{d_1 : l_1}} \mathsf{F}(0, \overrightarrow{X}) \mid \overrightarrow{C_1 : \mathcal{R}_1} & \dots \\
\textbf{where}\, i = j + 1 & \mathsf{K}'_1 : \quad \overrightarrow{B'_1 : \mathcal{T}'_1} \vdash_{\overrightarrow{d'_1 : l'_1}} \mathsf{F}(j+1, \overrightarrow{X}) \mid \overrightarrow{C'_1 : \mathcal{R}'_1} & \dots
\end{array}$$

the $i = 0$ case proceeds as normal for a non-recursive data declaration without i in scope, but $i = j + 1$ case we can allow for the extra rule stating that $\mathsf{F}(j, \overrightarrow{A}) : \mathcal{S}$ for any $\overrightarrow{A : k}$.

$$\frac{\overrightarrow{\Theta, j : \mathsf{Ix}, \Theta' \vdash_{\mathcal{G}} A : k}}{\Theta, j : \mathsf{Ix}, \Theta' \vdash_{\mathcal{G}} \mathsf{F}(j, \overrightarrow{A}) : \mathcal{S}}$$

Intuitively, in the $i = j + 1$ case the sequents for the constructors may additionally refer to smaller instances $\mathsf{F}(j, \overrightarrow{A})$ of the type being defined. This gives us the following rule for primitive recursive data declarations:

$$\dfrac{\begin{array}{c}\mathcal{G} \vdash\ \textbf{data}\, \mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\, \textbf{where} \\ \overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y : l}} \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B : \mathcal{R}}\right)}\end{array} \qquad \begin{array}{c} \dfrac{\overrightarrow{\Theta, j : \mathsf{Ix}, \Theta' \vdash_{\mathcal{G}} A : k}}{\Theta, j : \mathsf{Ix}, \Theta' \vdash_{\mathcal{G}} \mathsf{F}(j, \overrightarrow{A}) : \mathcal{S}} \\ \vdots \\ \mathcal{G} \vdash\ \textbf{data}\, \mathsf{F}(j : \mathsf{Ix}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{where} \\ \overrightarrow{\mathsf{K}' : \left(\overrightarrow{A' : \mathcal{T}'} \vdash^{\overrightarrow{Y : l}} \mathsf{F}(j, \overrightarrow{X}) \mid \overrightarrow{B' : \mathcal{R}'}\right)}\end{array}}{\begin{array}{ll} \multicolumn{2}{l}{\mathcal{G} \vdash\ \textbf{data}\, \mathsf{F}(i : \mathsf{Ix}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{by}\, \text{primitive recursion on } i} \\ \textbf{where}\, i = 0 & \overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y : l}} \mathsf{F}(0, \overrightarrow{X}) \mid \overrightarrow{B : \mathcal{R}}\right)} \\ \textbf{where}\, i = j + 1 & \overrightarrow{\mathsf{K}' : \left(\overrightarrow{A' : \mathcal{T}'} \vdash^{\overrightarrow{Y' : l'}} \mathsf{F}(j+1, \overrightarrow{X}) \mid \overrightarrow{B' : \mathcal{R}'}\right)} \end{array}}\ \mathit{data}_{\mathit{prim}}$$

And well-formedness of primitive recursive co-data types are the same. If the declaration is well-formed, we have the typing rules for F similarly to a non-recursive (co-)data type. The difference is that the (co-)constructors for the $i = 0$ and $i = j + 1$ case build a structure of type $\mathsf{F}(0, \overrightarrow{A})$ and $\mathsf{F}(M + 1, \overrightarrow{A})$ with M substituted for j, respectively. Additionally, there are two case abstractions: one of type $\mathsf{F}(0, \overrightarrow{A})$ that only handles constructors of the $i = 0$ case, and one of type $\mathsf{F}(M + 1, \vec{A})$ that only handles constructors of the $i = j + 1$ case.

Similarly, when checking for well-formedness of (co-)data types $\mathsf{F}(i : \mathsf{Ord}, \overrightarrow{X : k})$ defined by noetherian induction on $i : \mathsf{Ord}$, we get to assume the type is defined for smaller indices:

$$\frac{\Theta, i : \mathsf{Ord}, \Theta' \vdash_{\mathcal{G}} M : < i \quad \overrightarrow{\Theta, i : \mathsf{Ord}, \Theta' \vdash_{\mathcal{G}} A : k}}{\Theta, i : \mathsf{Ord}, \Theta' \vdash_{\mathcal{G}} \mathsf{F}(M, \vec{A}) : \mathcal{S}}$$

Intuitively, the sequents for the constructors may refer to $\mathsf{F}(M, \vec{A})$, so long as they introduce quantified type variables $\overrightarrow{Y : l}$ such that $\overrightarrow{X : k}, \overrightarrow{Y : l} \vdash M < i$. Other than this, the typing rules for structures and case statements are exactly the same as for non-recursive (co-)data types. This gives us the following rule for noetherian recursive data declarations:

$$\dfrac{\begin{array}{c} \dfrac{\Theta, i : \mathsf{Ord}, \Theta' \vdash_{\mathcal{G}} M : < i \quad \overrightarrow{\Theta, i : \mathsf{Ord}, \Theta' \vdash_{\mathcal{G}} A : k}}{\Theta, i : \mathsf{Ord}, \Theta' \vdash_{\mathcal{G}} \mathsf{F}(M, \vec{A}) : \mathcal{S}} \\ \vdots \\ \mathcal{G} \vdash \ \textbf{data}\, \mathsf{F}(i : \mathsf{Ord}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{where} \\ \qquad\qquad \mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(i, \vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right) \end{array}}{\begin{array}{l} \mathcal{G} \vdash \ \textbf{data}\, \mathsf{F}(i : \mathsf{Ord}, \overrightarrow{X : k}) : \mathcal{S}\, \textbf{by}\, \text{noetherian recursion on } i\, \textbf{where} \\ \qquad\qquad \mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(i, \vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right) \end{array}} \; data_{noether}$$

And well-formedness of noetherian recursive co-data types are the same. If the declaration is well-formed, we have exactly the typing rules for F as non-recursive (co-)data type.

Having concluded the static semantics of well-founded recursion in the sequent calculus, we also need to explain the impact on the dynamic semantics. In particular, there are the two dual pairs of special case abstractions introduced in Figure 6.8 that allow for self-reference. To compute with recursion, we use the additional rules shown in Figure 6.10. The recursive case abstractions for $\forall_{<\mathsf{Ord}}$ and $\exists_{<\mathsf{Ord}}$ are simplified by "unrolling" their loop via the ν rules: the recursive abstraction reduces to a non-recursive one by substituting itself inward—with a tighter upper bound—for the recursive variable. Intuitively, this index-unaware loop unrolling is possible because the actual chosen index *doesn't matter*, the loop must do the same thing each time

$$
\begin{aligned}
(\nu) &\qquad \mu([j{<}N \mathbin{@_x} \alpha].c) \succ_\nu \mu([i{<}N \mathbin{@} \alpha].c\,\{i/j, \mu([j{<}i \mathbin{@_x} \alpha].c)/x\}) \\
(\nu) &\qquad \tilde{\mu}[(j{<}N \mathbin{@_\alpha} x).c] \succ_\nu \tilde{\mu}[(i{<}N \mathbin{@} x).c\,\{i/j, \tilde{\mu}[(j{<}i \mathbin{@_\alpha} x).c]/\alpha\}]
\end{aligned}
$$

$$
\begin{aligned}
(\beta_{\mathcal{S}}^{\forall_{\mathsf{Ix}}}) &\qquad \langle V \| 0 \mathbin{@} E\rangle \succ_{\beta_{\mathcal{S}}^{\forall_{\mathsf{Ix}}}} c_0\,\{E/\alpha\} \\
(\beta_{\mathcal{S}}^{\forall_{\mathsf{Ix}}}) &\qquad \langle V \| M{+}1 \mathbin{@} E\rangle \succ_{\beta_{\mathcal{S}}^{\forall_{\mathsf{Ix}}}} \langle \mu\beta.\,\langle V \| M \mathbin{@} \beta\rangle \| \tilde{\mu}x.c_1\,\{M/j, E/\alpha\}\rangle \\
&\qquad \textbf{where}\, V = \mu([0{:}\mathsf{Ix} \mathbin{@} \alpha].c_0 \mid [j{+}1{:}\mathsf{Ix} \mathbin{@_x} \beta].c_1) \\
(\beta_{\mathcal{S}}^{\exists_{\mathsf{Ix}}}) &\qquad \langle 0 \mathbin{@} V \| E\rangle \succ_{\beta_{\mathcal{S}}^{\exists_{\mathsf{Ix}}}} c_0\,\{V/x\} \\
(\beta_{\mathcal{S}}^{\exists_{\mathsf{Ix}}}) &\qquad \langle M{+}1 \mathbin{@} V \| E\rangle \succ_{\beta_{\mathcal{S}}^{\exists_{\mathsf{Ix}}}} \langle \mu\alpha.c_1\,\{M/j, V/y\} \| \tilde{\mu}y.\,\langle M \mathbin{@} y \| E\rangle\rangle \\
&\qquad \textbf{where}\, E = \tilde{\mu}[(0{:}\mathsf{Ix} \mathbin{@} x).c_0 \mid (j{+}1{:}\mathsf{Ix} \mathbin{@_\alpha} y).c_1]
\end{aligned}
$$

FIGURE 6.10. Rewriting theory for recursion in the parametric $\mu\tilde{\mu}$-calculus.

around regardless of the value of the index. In contrast, the $\forall_{\mathsf{Ix}}$ and $\exists_{\mathsf{Ix}}$ recursors operate strictly stepwise: they will always go from step 10 to 9 and so on to 0. The indices used in the constructor really do matter, because they can influence the behavior of the program. This fact forces us to "unroll" the loop while pattern-matching on structures like $(M{+}1) @ E$ in tandem; unlike noetherian recursion the two steps cannot be performed independently.

The "well-foundedness" of the recursion corresponds to strong normalization of the rewriting theory: every possible reduction sequence is finite. Unfortunately, even with the indexes controlling the use of self-reference, the naïve reduction theory allows for infinite reduction sequences, albeit pointless ones. In particular, notice how the ν rules from Figure 6.10 are self-replication: so long as the self-referential variable x (in the case of $\forall_{<\mathsf{Ord}}$) or α (in the case of $\exists_{<\mathsf{Ord}}$) occur inside the command c, then the result of a ν reduction contains yet another self-referential abstraction that can be unrolled again. Therefore, in the interest of strong normalization, we must impose a restriction on where reduction can occur to prevent these pointless infinite unrollings. Intuitively, the restriction should follow the motto "don't touch unreachable branches." In other words, we limit the compatible closure of the rewriting rules, which normally allows reduction to occur in any context, to be careful about reduction inside a case abstraction.

The simplest such restriction is to use the *weak* reduction theory, which allows for reduction inside any context *except* inside of case abstractions. This corresponds to weak reduction in the λ-calculus which does not allow for reduction inside of λ-abstractions. While weak reduction suffices for strong normalization, it can be a bit too draconian. Therefore, we consider something in between general reduction and weak reduction that allows for some reduction inside case abstractions, but only those that satisfy a *reachability* caveat about the kinds of quantified types introduced by its patterns. This restriction prevents unnecessary infinite unrolling that would otherwise occur in simple commands like $\left\langle \mathit{length} \middle\| \mathsf{Rise}^{i}[\alpha] \right\rangle$. Intuitively, the reachability caveat prevents reduction inside a case abstraction which introduces type variables that might be *impossible* to instantiate, like $i < 0$ or $j < i$. The compatible reductions following the reachability caveat are defined as:

$$\frac{c \to c' \quad b : \overrightarrow{k \to}(< N) \in \Theta \implies N = \infty \vee N = M + 1}{\mu\Big(\mathsf{O}^{\Theta}[\vec{x}, \vec{\alpha}].c | \dots\Big) \to \mu\Big(\mathsf{O}^{\Theta}[\vec{x}, \vec{\alpha}].c' | \dots\Big)}$$

$$\frac{c \to c' \quad b : \overrightarrow{k \to}(< N) \in \Theta \implies N = \infty \vee N = M + 1}{\tilde{\mu}\Big[\mathsf{K}^{\Theta}(\vec{\alpha}, \vec{x}).c | \dots\Big] \to \tilde{\mu}\Big[\mathsf{K}^{\Theta}(\vec{\alpha}, \vec{x}).c' | \dots\Big]}$$

We call the reduction theory which follows the above caveats the *bounded* reduction theory, and note that every weak reduction step is a bounded reduction step, which is in turn a general reduction step.

We also define the type erasure operation on programs, $\mathit{Erase}(c)$, $\mathit{Erase}(v)$, and $\mathit{Erase}(e)$ as shown in Figure 6.11, which removes all types from constructors and patterns in c with an erasable kind, while leaving intact the unerasable Ix types. The corresponding type-erased reduction theory is the same, except we can no longer rely on the reachability caveat to maintain strong normalization while reducing inside of case abstractions. Instead, in the type-erased calculus, we must assume the worst and can only use the weak reduction theory if we want strong normalization to hold without the help of the upper bounds on noetherian recursion. In this case every weak reduction step of a type-erased command is justified by the same weak reduction step in the original command, so that type-erasure cannot introduce infinite loops.

Since the reduction theory of the $\mu\tilde{\mu}$-calculus is parameterized by a particular (substitution) strategy $\mathcal{S}$, the strong normalization of well-founded also follows by

$$
\begin{aligned}
\mathit{Erase}\langle v \| e\rangle &\triangleq \langle \mathit{Erase}(v) \| \mathit{Erase}[e]\rangle \\[1ex]
\mathit{Erase}(x) &\triangleq x \\
\mathit{Erase}(\mu\alpha.c) &\triangleq \mu\alpha.\mathit{Erase}(c) \\
\mathit{Erase}(\mathsf{K}^{\vec{A}}(\vec{e}, \vec{v})) &\triangleq \mathsf{K}^{\mathit{Erase}(\vec{A})}(\overrightarrow{\mathit{Erase}[e]}, \overrightarrow{\mathit{Erase}(v)}) \\
\mathit{Erase}\Big(\mu\Big(\overrightarrow{\mathsf{O}^{\overrightarrow{Y{:}l}}[\vec{x}, \vec{\alpha}].c}\Big)\Big) &\triangleq \mu\Big(\overrightarrow{\mathsf{O}^{\mathit{Erase}(\overrightarrow{Y{:}l})}[\vec{x}, \vec{\alpha}].\mathit{Erase}(c)}\Big) \\[1ex]
\mathit{Erase}[\alpha] &\triangleq \alpha \\
\mathit{Erase}[\tilde{\mu}x.c] &\triangleq \tilde{\mu}x.\mathit{Erase}(c) \\
\mathit{Erase}[\mathsf{O}^{\vec{A}}[\vec{v}, \vec{e}]] &\triangleq \mathsf{O}^{\mathit{Erase}(\vec{A})}(\overrightarrow{\mathit{Erase}(v)}, \overrightarrow{\mathit{Erase}[e]}) \\
\mathit{Erase}\Big[\tilde{\mu}\Big[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{\alpha}, \vec{x}).c}\Big]\Big] &\triangleq \tilde{\mu}\Big[\overrightarrow{\mathsf{K}^{\mathit{Erase}(\overrightarrow{Y{:}l})}(\vec{\alpha}, \vec{x}).\mathit{Erase}(c)}\Big] \\[1ex]
\mathit{Erase}(\epsilon) &\triangleq \epsilon \\
\mathit{Erase}(A : k, \overrightarrow{B : k}) &\triangleq \mathit{Erase}(\overrightarrow{B : k}) && \textbf{if } k : \square \\
\mathit{Erase}(A : k, \overrightarrow{B : k}) &\triangleq A : k, \mathit{Erase}(\overrightarrow{B : k}) && \textbf{if } k : \blacksquare
\end{aligned}
$$

FIGURE 6.11. Type erasure for the higher-order parametric $\mu\tilde{\mu}$-calculus.

an argument that is parametric with respect to $\mathcal{S}$. It doesn't matter exactly which (co-)values are substitutable, but only that $\mathcal{S}$ meets some general conditions. In particular, strong normalization of the $\beta\varsigma\nu$ reduction theory follows for any $\mathcal{S}$ which is *focalizing* (as in Definition 5.2 discussed in Chapter V Section 5.3) and *stable*.

Definition 6.1 (Stable strategy)**.** A substitution strategy $\mathcal{S}$ is *stable* if and only if (co-)values are closed under reduction.

It then follows that the bounded reduction theory is strongly normalizing on well-typed commands and (co-)terms for any stable and focalizing $\mathcal{S}$, and therefore so is the weak reduction theory on both higher-order and type-erased programs.

Theorem 6.1. *Suppose that $\mathcal{S}$ is a stable and focalizing substitution strategy.*

a) If $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ and $\left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}$, then the bounded $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in c and the weak $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in Erase(c).

b) If $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta$, $\Theta \vdash_{\mathcal{G}} A : \mathcal{S}$, and $\left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}$, then the bounded $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in v and the weak $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in Erase(v).

c) If $\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta$, $\Theta \vdash_{\mathcal{G}} A : \mathcal{S}$, and $\left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}$, then the bounded $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in e and the weak $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in Erase(e).

Note that the call-by-value ($\mathcal{V}$), call-by-name ($\mathcal{N}$), call-by-need ($\mathcal{LV}$) and its dual ($\mathcal{LN}$) from Section 5.1 are all stable and focalizing, so that as a corollary, we achieve strong normalization for these particular instances of the parametric $\mu\tilde{\mu}$-calculus. Furthermore, the non-deterministic strategy $\mathcal{U}$ is also stable and focalizing, which gives another account of strong normalization for the symmetric λ-calculus (Lengrand & Miquel, 2008) as a corollary and shows its extension to other programming features including recursion. The details for this proving strong normalization are given in the extended version of (Downen *et al.*, 2015).

Encoding Recursive Programs via Structures

To see how to encode basic recursive definitions into the sequent calculus using the primitive and noetherian recursion principles, we revisit the previous examples from Section 6.1 to show how to encode basic recursive definitions into values in the sequent calculus. For simplicity, we will stick to a single generic base kind $\mathcal{S}$, although each example can be adapted to use multiple, like the polarized mixture of $\mathcal{V}$ and $\mathcal{N}$, as desired. We will see how the intuitive argument for termination can be represented using the type indices for recursion in various ways. In essence, we demonstrate how the parametric $\mu\tilde{\mu}_{\mathcal{S}}$-calculus can be used as a core calculus and compilation target for establishing well-foundedness of recursive programs.

Example 6.5. Recall the *length* function from Example 6.1, as written in sequent-style. As we saw, we could internalize the definition for *length* into a recursively-defined case abstraction that describes each possible behavior. Using the noetherian recursion principle in the $\mu\tilde{\mu}$-calculus, we can give a more precise and non-recursive definition for *length*:

$$\begin{aligned}
&\mathit{length} : \forall X : \mathcal{S}.\forall i < \infty.\, \mathsf{List}(i, X) \to \mathsf{Nat}(i) \\
&\mathit{length} = \mu([X \mathbin{@} i{<}\infty \mathbin{@}_r \mathsf{Nil} \cdot \gamma].\langle \mathsf{Z} \| \gamma \rangle \\
&\qquad\qquad |[X \mathbin{@} i{<}\infty \mathbin{@}_r \mathsf{Cons}^{j<i}(x, xs) \cdot \gamma].\left\langle r \middle\| j \mathbin{@} xs \cdot \tilde{\mu} y.\left\langle \mathsf{S}^j(y) \middle\| \gamma \right\rangle \right\rangle)
\end{aligned}$$

The difference is that the polymorphic nature of the *length* function is made explicit in system F style, and the recursion part of the function has been made internal through the $\forall_{<\mathsf{Ord}}$ co-data type. Going further, we may unravel the deep patterns into shallow case analysis, giving annotations on the introduction of every co-variable:

$$\begin{aligned}
\mathit{length} = \mu(&[X \mathbin{@} \alpha{:}\forall i < \infty.\, \mathsf{List}(i, X) \to \mathsf{Nat}(i)]. \\
&\langle \mu([i{<}\infty \mathbin{@}_{r:\forall j<i.\,\mathsf{List}(j,X)\to\mathsf{Nat}(j)} \beta{:}\, \mathsf{List}(i, X) \to \mathsf{Nat}(i)]. \\
&\quad \langle \mu([xs{:}\mathsf{List}(i, a) \cdot \gamma{:}\mathsf{Nat}(i)]. \\
&\qquad \langle xs \| \tilde{\mu}[\mathsf{Nil}.\langle \mathsf{Z} \| \gamma \rangle \\
&\qquad\qquad |\mathsf{Cons}^{j<i}(x{:}X, ys{:}\mathsf{List}(j, X)).\langle r \| j \mathbin{@} ys \cdot \tilde{\mu} y{:}\, \mathsf{Nat}(j).\langle \mathsf{S}^j(y) \| \gamma \rangle\rangle])| \\
&\quad |\beta\rangle)| \\
&|\alpha\rangle)
\end{aligned}$$

Although quite verbose, this definition spells out all the information we need to verify that *length* is well-typed and well-founded: no guessing required. Furthermore, this core definition of *length* is entirely in terms of shallow case analysis, making reduction straightforward to implement. Since the correctness of programs is ensured for this core form, which can be elaborated from the deep pattern-matching definition mechanically, we will favor the more concise pattern-matching forms for simplicity in the remaining examples. *End example* 6.5.

Example 6.6. Recall the *countUp* function from Example 6.2. When we attempt to encode this function into the parametric $\mu\tilde{\mu}$-calculus, we run into a new problem: the indices for the given number and the resulting stream do not line up since one grows while the other shrinks. To get around this issue, we mask the index of the given natural number using the dual form of noetherian recursion, and say that $\mathsf{ANat} = \exists i{<}\infty.\, \mathsf{Nat}(i)$. We can then describe *countUp* as a function from ANat to a $\mathsf{Stream}(i, \mathsf{ANat})$ by noetherian recursion on i:

$$\begin{aligned}
&\mathit{countUp} : \forall i < \infty.\, \mathsf{ANat} \to \mathsf{Stream}(i, \mathsf{ANat}) \\
&\mathit{countUp} = \mu([i{<}\infty \mathrel{@_r} x \cdot \mathsf{Head}[\alpha]].\langle x \| \alpha \rangle \\
&\qquad\qquad\qquad |[i{<}\infty \mathrel{@_r} (j{<}i \mathrel{@} x) \cdot \mathsf{Tail}^{k<i}[\beta]].\langle r \| k \mathrel{@} (j{+}1 \mathrel{@} \mathsf{S}^j(x)) \cdot \beta \rangle)
\end{aligned}$$

End example 6.6.

Example 6.7. The previous example shows how infinite streams may be modeled by co-data. However, recall the other approach to infinite objects mentioned in Example 6.4. Unfortunately, an infinitely constructed list like *zeroes* would be impossible to define in terms of noetherian recursion: in order to use the recursive argument, we need to come up with an index smaller than the one we are given, but since lists are a data type their observations are inscrutable and we have no place to look for one. As it turns out, though, primitive recursion is set up in such a way that we can make headway. Defining infinite lists to be $\mathsf{InfList}(X) = \forall i : \mathsf{Ix}.\, \mathsf{IxList}(i, X)$, we can encode *zeroes* as:

$$\begin{aligned}
&\mathit{zeroes} : \mathsf{InfList}(\mathsf{Nat}(0)) \\
&\mathit{zeroes} = \mu([0{:}\mathsf{Ix} \mathrel{@} \alpha{:}\, \mathsf{IxList}(0, \mathsf{Nat}(0))].\langle \mathsf{Nil} \| \alpha \rangle \\
&\qquad\qquad |[i{+}1{:}\mathsf{Ix} \mathrel{@_{r:\, \mathsf{IxList}(i, \mathsf{Nat}(0))}} \alpha{:}\, \mathsf{IxList}(i+1, \mathsf{Nat}(0))].\langle \mathsf{Cons}(\mathsf{Z}, r) \| \alpha \rangle)
\end{aligned}$$

Even more, we can define the concatenation of infinitely constructed lists in terms of primitive recursion as well. We give a wrapper, *cat*, that matches the indices of the incoming and outgoing list structure, and a worker, *cat*′, that performs the actual recursion:

$$\begin{aligned}
&\mathit{cat} : \forall X : \mathcal{S}.\, \mathsf{InfList}(X) \to \mathsf{InfList}(X) \to \mathsf{InfList}(X) \\
&\mathit{cat} = \langle \mu([X{:}\mathcal{S} \mathbin{@} xs \cdot ys \cdot i{:}\mathsf{Ix} \mathbin{@} \alpha].\langle xs \| i \mathbin{@} \tilde{\mu} zs.\, \langle \mathit{cat}' \| i \mathbin{@} zs \cdot ys \cdot \alpha \rangle \rangle)
\end{aligned}$$

$$\begin{aligned}
\mathit{cat}' &: \forall X : \mathcal{S}. \forall i : \mathsf{Ix}.\, \mathsf{IxList}(i, X) \to \mathsf{InfList}(X) \to \mathsf{IxList}(i, X) \\
\mathit{cat}' &= \mu([X \mathbin{@} 0 \mathbin{@} \mathsf{Nil} \cdot ys \cdot \alpha].\langle \mathsf{Nil} \| \alpha \rangle \\
&\qquad |[X \mathbin{@} i{+}1 \mathbin{@_r} \mathsf{Nil} \cdot ys \cdot \alpha].\langle ys \| i{+}1 \mathbin{@} \alpha \rangle \\
&\qquad |[X \mathbin{@} i{+}1 \mathbin{@_r} \mathsf{Cons}(x, xs) \cdot ys \cdot \alpha].\langle \mathsf{Cons}(x, \mu\beta.\, \langle r \| xs \cdot ys \cdot \beta \rangle) \| \alpha \rangle)
\end{aligned}$$

If we would like to stick with the "finite objects are data, infinite objects are co-data" mantra, we can write a similar concatenation function over possibly terminating streams:

$$\begin{aligned}
&\mathbf{codata}\, \mathsf{StopStream}(i < \infty, X : \mathcal{S}) : \mathcal{S}\, \mathbf{where} \\
&\quad \mathsf{Head} : \ \mid \mathsf{StopStream}(i, X) \vdash X : \mathcal{S} \\
&\quad \ \mathsf{Tail} : \ \mid \mathsf{StopStream}(i, X) \vdash^{j<i} 1 : \mathcal{S}, \mathsf{StopStream}(j, X) : \mathcal{S}
\end{aligned}$$

A $\mathsf{StopStream}(i, X)$ object is like a $\mathsf{Stream}(i, X)$ object except that asking for its Tail might fail and return the unit value instead, so it represents an infinite or finite stream of one or more values. This co-data type makes essential use of multiple conclusions, which are only available in a language for classical logic. We can now write a general recursive definition of concatenation in terms of the $\mathsf{StopStream}$ co-data type:

$$\begin{aligned}
\langle \mathit{cat} \| xs \cdot ys \cdot \mathsf{Head}[\alpha] \rangle &= \langle xs \| \mathsf{Head}[\alpha] \rangle \\
\langle \mathit{cat} \| xs \cdot ys \cdot \mathsf{Tail}[\delta, \beta] \rangle &= \langle \mathit{cat} \| \mu\gamma.\, \langle xs \| \mathsf{Tail}[\tilde{\mu}[()].\langle ys \| \beta \rangle] \,, \gamma] \rangle \cdot ys \cdot \beta \rangle
\end{aligned}$$

This function encodes into a similar pair of worker-wrapper values, where now a possibly infinite list is represented as a terminating stream $\mathsf{InfList}(X) = \forall i < \infty.\, \mathsf{StopStream}(i, X)$:

$$\begin{aligned}
\mathit{cat}' &: \forall X : \mathcal{S}. \forall i < \infty.\, \mathsf{StopStream}(i, X) \to \mathsf{InfList}(X) \to \mathsf{StopStream}(i, X) \\
\mathit{cat}' &= \mu([X \mathbin{@} i{<}\infty \mathbin{@_r} xs \cdot ys \cdot \mathsf{Head}[\alpha]].\langle xs \| \alpha \rangle \\
&\qquad |[X \mathbin{@} i{<}\infty \mathbin{@_r} xs \cdot ys \cdot \mathsf{Tail}^{j<i}[\delta, \beta]].
\end{aligned}$$

$$\langle r \| j \mathbin{@} \mu\gamma.\langle xs \| \,\mathsf{Tail}^j[\tilde{\mu}[().\langle ys \| j \mathbin{@} \beta\rangle]\,, \gamma]\rangle \cdot ys \cdot \beta\rangle)$$

End example 6.7.

Remark 6.1. It is worth pointing out why our encoding for “infinite” data structures, like *zeroes*, avoids the problem underlying the lack of subject reduction for co-induction in Coq (Oury, 2008). Intuitively, the root of the problem is that Coq’s co-inductive objects are non-extensional, since the interaction between case analysis and the co-fixed point operator effectively allows these objects to notice if they are being discriminated or not. In contrast, we take the extensional view that the presence or absence of case analysis, in *all* of its various forms, is unobservable. To ensure strong normalization, the basic observation is instead a specific message that advertises to the object exactly how deep it would like to go, thus restoring extensionality and putting a limit on unfolding. *End remark* 6.1.

Example 6.8. We now consider an example with a more complex recursive argument that makes non-trivial use of lexicographic induction. The Ackermann function can be written as:

$$\begin{aligned}
\langle ack \| \mathsf{Z} \cdot y \cdot \alpha\rangle &= \langle \mathsf{S}(y) \| \alpha\rangle \\
\langle ack \| \mathsf{S}(x) \cdot \mathsf{Z} \cdot \alpha\rangle &= \langle ack \| x \cdot \mathsf{S}(\mathsf{Z}) \cdot \alpha\rangle \\
\langle ack \| \mathsf{S}(x) \cdot \mathsf{S}(y) \cdot \alpha\rangle &= \langle ack \| \mathsf{S}(x) \cdot y \cdot \tilde{\mu}z.\, \langle ack \| x \cdot z \cdot \alpha\rangle\rangle
\end{aligned}$$

The fact that this function terminates follows by lexicographic induction on both arguments: to every recursive call of *ack*, either the first number decreases, or the first number stays the same and the second number decreases. This argument can be encoded into the basic noetherian recursion principle we already have by nesting it twice:

$$\begin{aligned}
&ack : \forall i < \infty. \forall j < \infty.\, \mathsf{Nat}(i) \to \mathsf{Nat}(j) \to \mathsf{ANat} \\
&ack = \mu([i{<}\infty \mathbin{@_{r_1}} j{<}\infty \mathbin{@_{r_2}} \mathsf{Z} \cdot y \cdot \alpha].\langle j{+}1 \mathbin{@} \mathsf{S}^j(y) \| \alpha\rangle \\
&\qquad\quad |[i{<}\infty \mathbin{@_{r_1}} j{<}\infty \mathbin{@_{r_2}} \mathsf{S}^{i'<i}(x) \cdot \mathsf{Z} \cdot \alpha].\langle r_1 \| i' \mathbin{@} 1 \mathbin{@} x \cdot \mathsf{S}^0(\mathsf{Z}) \cdot \alpha\rangle \\
&\qquad\quad |[i{<}\infty \mathbin{@_{r_1}} j{<}\infty \mathbin{@_{r_2}} \mathsf{S}^{i'<i}(x) \cdot \mathsf{S}^{j'<j}(y) \cdot \alpha]. \\
&\qquad\qquad \langle r_2 \| j' \mathbin{@} \mathsf{S}^{i'}(x) \cdot y \cdot \tilde{\mu}[(k{<}\infty \mathbin{@} z).\langle r_1 \| i' \mathbin{@} k \mathbin{@} x \cdot z \cdot \alpha\rangle]\rangle)
\end{aligned}$$

Essentially, we get two recursive arguments from nesting $\forall_{<\mathsf{Ord}}$ quantification over Ord indexes:

$$
\begin{aligned}
&r_1 : \forall i' < i. \forall j < \infty.\, \mathsf{Nat}(i') \to \mathsf{Nat}(j) \to \mathsf{ANat}\\
&r_2 : \forall j' < j.\, \mathsf{Nat}(i) \to \mathsf{Nat}(j') \to \mathsf{ANat}
\end{aligned}
$$

The first recursive path r_1 can be taken whenever the first argument is smaller, in which case the second argument is arbitrary. The second recursive path r_2 can be taken whenever the second argument is smaller and the first argument has the same index (the i in the type of r_2 matches the index of the original first argument to *ack*). Again, we find that the dual form noetherian recursion, $\exists_{<\mathsf{Ord}}$, is useful for masking the index of the output from *ack*. Furthermore, it is interesting to note that in the third case of *ack*, we must explicitly destruct the $\exists$-packed result from *ack* before performing the second recursive call. In practical terms, this forces the nested recursive call of the Ackermann function to be strict, even in a lazy language. *End example* 6.8.

Each of these examples shows how we can phrase many different inductive and co-inductive arguments in the form of structural recursion on combinations of data and co-data types, where the forms of structural recursion provided by the calculus are guaranteed to be well-founded by the strong normalization theorem (Theorem 6.1). The next Chapter VII will present a parametric model for the parametric sequent calculus which is suitable for proving strong normalization. The model in Chapter VII is simplified from the one used in (Downen *et al.*, 2015) which has both a positive and negative consequence. An unfortunate cost of the simplification is that the model in Chapter VII only applies to deterministic strategies like $\mathcal{V}$, $\mathcal{N}$, $\mathcal{LV}$, $\mathcal{LN}$, or the polarized $\mathcal{P}$ which resolve the fundamental dilemma of classical computation, and do not accomodate the type of non-determinism allowed by the $\mathcal{U}$ substitution strategy. However, a technical benefit of the simplification is that the model in Chapter VII straightforwardly scale to tackle other problems unrelated to strong normalization, like the question of whether the typed $\beta\eta$ theory of (co-)data developed previously in Chapter V is “sound” with respect to the untyped $\beta\varsigma$ theory in some appropriate sense.

CHAPTER VII

Parametric Orthogonality Models

We have now studied many languages for the sequent calculus which include features like simple, higher-order, and recursive types framed as data and co-data. This study has included both the static (i.e. type systems) and dynamic (i.e. rewriting rules) semantics for the features in question. The dynamic semantics in particular was done in two different styles: one typed $\beta\eta$ system for determining when programs using (co-)data are equal, and one untyped $\beta\varsigma$ system for running a program to find the answer. As we saw in Section 5.3 of Chapter V the two different versions of the dynamic semantics are related: $\beta\eta$ subsumes $\beta\varsigma$ for typed commands, terms, and co-terms (Theorem 5.2). However, what about the other direction? The meaning of a program should be defined by how it behaves when it is run. So if we are ultimately using $\beta\varsigma$ to evaluate (co-)data structures, then where does that leave the extensional η law? Is ς truly the operational shadow of η that is seen during execution, or does η mean something else?

This problem is confounded by the fact that purely syntactic methods are not easy to apply to simplify η down to the operational semantics. That's because the η law is often just *too strong* and breaks any obvious form of confluence, defeating any syntactic techniques based on it. Therefore, if we want to reconcile the strong η rule with the operational semantics, we need to employ a different approach. For that reason, we move to a semantic methodology that builds a *model* for the language, and show that the syntactic typing rules adequately reflect the meaning of programs in the semantic model. In turn, the model lets us capture more challenging properties of the language, like the fact that the untyped operational semantics respects the typed extensional η law. Taken together, these two parts bridge the static and dynamic semantics of the language, and let us make bolder claims.

This chapter incorporates ideas from the proof technique for strong normalization from the supporting material in the appendix of (Downen *et al.*, 2015) which was developed in collaboration with Philip Johnson-Freyd.

We now seek to build a model for the parametric $\mu\tilde{\mu}$-calculus that lets us capture the impact of types on the way that programs run. The model that we build is parameterized by a number of different choices:

1. a *safety condition* constraining the run-time behavior of programs, so that well-typed programs don't "go wrong,"
2. a collection of (co-)data type declarations that define what types are interpreted as semantic entities of the model, and
3. a collection of evaluation strategies that define how programs are executed.

The first parameter is not too uncommon—there are models of languages that abstract out a notion of "safety," which lets them speak about many different properties of the language all at once. However, the second two parameters are novel—usually, models consider a language with a fixed set of type formers and a fixed evaluation mechanism—which is due to the open-ended presentation of (co-)data and evaluation in Chapter V.

Our parameterized model will be based on an idea by Girard (1987) which goes by many names: (bi-)orthogonality (Munch-Maccagnoni, 2009), classical realizability Krivine (2009), and $\top\top$-closure Pitts (2000). The basic idea is to capture safety as a binary predicate ($\perp\!\!\!\perp$) on two opposite entities: answers and questions. We can pose the two dual problems: "which questions are safe to ask about these answers?" and "which answers are safe to give to these questions?" This style of formulation matches perfectly with the language of the sequent calculus. Terms are answers, co-terms are questions, and commands are the action of asking a particular question about a particular answer. The $\perp\!\!\!\perp$ predicate represents a collection of commands that are safe to run. The safety properties of types are then modeled by a collection of answers (i.e. terms) and questions (i.e. co-terms) where every possible questions-answers combination is safe (i.e. a command in $\perp\!\!\!\perp$). The orthogonality approach gives a heavily test-based view of language properties, where we use test suits of canonical observations to carve out a space of valid programs that pass the test for each of those observations, or alternative a specification of obviously correct results to carve out a space of valid use-cases. The magic of this approach is that we quickly reach a fixed point: after flipping back and forth between questions and answers with orthogonality twice, we learn everything we possibly can.

This chapter covers the following topics:

- A general introduction to the idea of orthogonality in an abstract setting of *spaces* and *poles* (Section 7.1) that explores the connection between orthogonality and negation in intuitionistic logic.
- A representation of types based on orthogonality that are oriented around either a positive or negative bias (Section 7.2), and a generic presentation of the *closure under expansion* property, which is pervasive to semantic models of programming languages, which is appropriate for many different applications.
- A binary model of the parametric $\mu\tilde{\mu}$-calculus with higher-order and recursive types that interprets sequents as statements about program behavior (Section 7.3), which is parameterized by a choice of (co-)data types, evaluation strategies, and safety condition.
- A proof of the fundamental *adequacy* lemmas (Section 7.4): the existence of a syntactic derivation of a sequent implies the truth of the semantic interpretation of that sequent.
- Several applications of the model, using adequacy to prove language-wide facts about the parametric $\mu\tilde{\mu}$-calculus (Section 7.5), including: logical consistency, type safety, strong normalization, and soundness of the extensional $\beta\eta$ theory with respect to the operational $\beta\varsigma$ semantics.

Poles, Spaces, and Orthogonality

We're going to look at a semantic model of understanding computation in the sequent calculus in terms of *orthogonality.* The model hinges on a representation of commands of the sequent calculus that we deem to be valid execution states for our purposes. In other words, we isolate some form of commands that can *run.* We represent such a set of runnable commands abstractly as a *computational pole* which is any set capable of running with a computation relation $\rightsquigarrow$. This way, the model is extensible and does not pin down the precise nature of commands ahead of time.

Definition 7.1 (Computational poles)**.** A *computational pole* $\mathbb{P}$ (or just *pole* for short) is any set equipped with a relation $\rightsquigarrow$ between elements of $\mathbb{P}$.

In addition, the terms and co-terms of the sequent calculus are represented as an *interaction space* with a positive and negative side oriented around some pole, which likewise abstracts over their precise form.

Definition 7.2 (Interaction spaces)**.** Given any computational pole $\mathbb{P}$, a $\mathbb{P}$-*interaction space* $\mathbb{A}$ (or just $\mathbb{P}$-*space* for short) is a pair of sets $(\mathbb{A}_+, \mathbb{A}_-)$ equipped with a *cut operation* $\langle \_ \| \_ \rangle : A_+ \to A_- \to \mathbb{P}$ (i.e. for all $v \in A_+$ and $e \in A_-$, $\langle v \| e \rangle \in \mathbb{P}$).

We call $\mathbb{P}$ the pole of $\mathbb{A}$, $\mathbb{A}_+$ the positive side of $\mathbb{A}$, $\mathbb{A}_-$ the negative side of $\mathbb{A}$, and use the shorthand $v \in \mathbb{A}$ to denote $v \in \mathbb{A}_+$ and $e \in \mathbb{A}$ to denote $e \in \mathbb{A}_-$.

Note that, while spaces and poles are quite abstract, we can always substitute the more concrete syntactic notions of the language of the sequent calculus for better intuition. For example, consider the (single-kinded) parametric $\mu\tilde{\mu}$-calculus from Chapter V. The set of untyped commands from Figure 5.7, *Command*, is a perfectly fine computational pole, since the untyped operational reductions $\mapsto_{\mu_\mathcal{S}\tilde{\mu}_\mathcal{S}\beta_\mathcal{S}\varsigma_\mathcal{S}}$ serve as a computational relation $\rightsquigarrow$ on commands. Likewise, the sets of untyped terms and co-terms from Figure 5.7, (*Term*, *CoTerm*), is a perfectly fine *Command*-interaction space, since we have the syntactic cut $\langle v \| e \rangle$ formation of commands. This follows from the intuition that commands are the primary computational entities of the sequent calculus, whereas terms and co-terms provide a space for possible interactions (via cuts) that lead to computations. We could just as well limit our attention to closed programs (those without any free variables) as does Munch-Maccagnoni (2009) by considering the set of closed, untyped commands as a pole along with the sets of closed, untyped (co-)terms as an interaction space. Furthermore, if we are instead interested in strong normalization, then we can start with the sets of all strongly normalizing, untyped (co-)terms as an all-encompassing interaction space. Therefore, the appropriate space required for modeling programs really depends on what sort of outcome we are looking to achieve.

Since interaction spaces are just a pair of sets, we can compare when one interaction space is contained inside another by considering the two pointwise: all the positive and negative elements of the contained space must also be positive and negative elements of the other space, respectively.

Definition 7.3 (Containment)**.** Given two $\mathbb{P}$-spaces $\mathbb{A} = (\mathbb{A}_+, \mathbb{A}_-)$ and $\mathbb{B} = (\mathbb{B}_+, \mathbb{B}_-)$, we say that $\mathbb{A}$ *is inside* $\mathbb{B}$ (written $\mathbb{A} \sqsubseteq \mathbb{B}$) if and only if $\mathbb{A}_+ \subseteq \mathbb{B}_+$ and $\mathbb{A}_- \subseteq \mathbb{B}_-$. Equivalently, we say that $\mathbb{B}$ *contains* $\mathbb{A}$ (written $\mathbb{B} \sqsupseteq \mathbb{A}$) if and only if $\mathbb{B}_+ \supseteq \mathbb{A}_+$ and $\mathbb{B}_- \supseteq \mathbb{A}_-$.

Containment lets us specify when one interaction space is made up of parts of another. For example, the set of terms and co-terms of type $A \to B$ is inside the set of untyped terms and co-terms, since every typed (co-)term is also an untyped (co-)term, but not vice versa. This relationship is important for setting up a large, encompassing space as an area of interest, wherein lie many smaller sub-spaces of interest.

We are now ready to tackle the most fundamental operation on interaction spaces: *orthogonality*. Intuitively, orthogonality lets us pare down a large interaction space which may include some undesired interactions by selecting only the parts which pass some chosen criteria. To start with, we begin with some "plausibly well-behaved" but overly-permissive computational pole $\mathbb{P}$ and $\mathbb{P}$-interaction space $\mathbb{A}$ which includes every interaction and computational behavior we might be interested in observing, but also may allow for undesired interactions and behaviors. From there, we select a sub-pole $\mathbb{Q}$ of $\mathbb{P}$ that serves as a safety condition and only includes the desired computational behavior that we are interested in, along with a sub-$\mathbb{P}$-space $\mathbb{C}$ contained in $\mathbb{A}$ which serves as a specification laying out a set of criteria for evaluating the safety of elements in $\mathbb{A}$. Together, $\mathbb{Q}$ and $\mathbb{C}$ can be seen as a *test suite* for performing quality control and determining which elements of $\mathbb{A}$ are acceptable: each positive element of $\mathbb{A}$ (intuitively, untested programs) must pass the $\mathbb{Q}$ test when paired with every negative element of $\mathbb{C}$ (intuitively, vetted use-cases), and dually each negative element of $\mathbb{A}$ (intuitively, untested use-cases) must pass the $\mathbb{Q}$ test when paired with every positive element of $\mathbb{C}$ (intuitively, vetted programs).[1]

Definition 7.4 (Orthogonality)**.** Let $\mathbb{P}$ be a pole, $\mathbb{Q} \subseteq \mathbb{P}$ be a sub-pole of $\mathbb{P}$, $\mathbb{A} = (\mathbb{A}_+, \mathbb{A}_-)$ be a $\mathbb{P}$-space, and $\mathbb{C} = (\mathbb{C}_+, \mathbb{C}_-) \sqsubseteq \mathbb{A}$ be a $\mathbb{P}$-space inside $\mathbb{A}$. The *positive* $\mathbb{Q}$-*orthogonal* of $\mathbb{C}_-$ inside $\mathbb{A}_+$, written $\mathbb{C}_-^{\mathbb{Q}_{\mathbb{A}_+}}$, consists of the positive elements of $\mathbb{A}$ that form a $\mathbb{Q}$ element when cut with every negative element of $\mathbb{C}$ and is defined as:

$$\mathbb{C}_-^{\mathbb{Q}_{\mathbb{A}_+}} \triangleq \{v \in \mathbb{A}_+ \mid \forall e \in \mathbb{C}_-, \langle v \| e \rangle \in \mathbb{Q}\}$$

Dually, the *negative* $\mathbb{Q}$-*orthogonal* of $\mathbb{C}_+$ inside $\mathbb{A}_-$, written $\mathbb{C}_+^{\mathbb{Q}_{\mathbb{A}_-}}$, consists of all negative elements of $\mathbb{A}$ that form a $\mathbb{Q}$ element when cut with every positive element of $\mathbb{C}$ and is defined as:

$$\mathbb{C}_+^{\mathbb{Q}_{\mathbb{A}_-}} \triangleq \{e \in \mathbb{A}_- \mid \forall v \in \mathbb{C}_+, \langle v \| e \rangle \in \mathbb{Q}\}$$

[1] Traditionally, these operations are referred to as either $\mathbb{C}^{\perp}$ or $\mathbb{C}^{\top}$, but here we use the generalized notation $\mathbb{C}^{\mathbb{Q}_{\mathbb{A}}}$ which lets us vary both the safety condition $\mathbb{Q}$ as well as the encompassing space $\mathbb{A}$ of all potential programs in consideration.

Taken together, the $\mathbb{Q}$-*orthogonal complement* of $\mathbb{C}$ inside $\mathbb{A}$, written $\mathbb{C}^{\mathbb{Q}_{\mathbb{A}}}$, is the $\mathbb{Q}$-space given by both the positive and negative $\mathbb{Q}$-orthogonals of $\mathbb{C}$ inside $\mathbb{A}$:

$$(\mathbb{C}_+, \mathbb{C}_-)^{\mathbb{Q}_{(\mathbb{A}_+, \mathbb{A}_-)}} \triangleq (\mathbb{C}_-^{\mathbb{Q}_{\mathbb{A}_+}}, \mathbb{C}_+^{\mathbb{Q}_{\mathbb{A}_-}})$$

Example 7.1. For example, suppose we are trying to reason about the execution of well-typed programs. In other words, we want to model type safety of the operational semantics. For an all-encompassing interaction space, we can consider all untyped (co-)terms $\mathbb{U} = (\mathit{Term}, \mathit{CoTerm})$, which is centered around the pole *Command* containing all untyped commands. We would then need to design a pole that is a subset of all untyped commands, $\perp\!\!\!\perp \subseteq \mathit{Command}$, representing type safety to contain all valid states of type-safe execution that eventually leads to an acceptable result, and excludes stuck states that are caused by type errors. For example, $\perp\!\!\!\perp$ would not include commands like $\langle \mathsf{True} \| 1 \cdot [] \rangle$, $\langle \mu(x \cdot \alpha.c) \| \tilde{\mu}[(x, y)\,.c] \rangle$, $\langle \iota_1\,(1) \| \tilde{\mu}[(x, y)\,.c] \rangle$, and $\langle \mu(x \cdot \alpha.c) \| \pi_1\,[\beta] \rangle$, since they are all stuck on irrecoverable miscommunications like missing case analysis or data/co-data mismatches. Instead, $\perp\!\!\!\perp$ would include valid states where we may not have enough information to take the next step, but execution could potentially continue if we learn more. These would be states where we are stuck on a free variable, like $\langle f \| 1 \cdot [] \rangle$ or $\langle z \| \tilde{\mu}[(x, y)\,.c] \rangle$, or on a free co-variable, like $\langle \mathsf{True} \| \alpha \rangle$ or $\langle \mu(x \cdot \alpha.c) \| \beta \rangle$, and correspond to the "final commands" from Chapters III and IV. To complete type safety pole $\perp\!\!\!\perp$, we should also ensure that a command that eventually reaches a valid state in some number of steps is also valid. That is, if c' is in $\perp\!\!\!\perp$ and $c \mapsto\!\!\!\!\!\rightarrow c'$ then c is also in $\perp\!\!\!\perp$. This is commonly referred to as "closure under expansion" and is found in similar models of program evaluation.

Now, we can consider what the orthogonality operations mean for the above description of our choice of the safety pole $\perp\!\!\!\perp$ beginning with every (co-)term in the all-encompassing *Command*-space $\mathbb{U}$. For instance, the negative orthogonal $\{()\}^{\perp\!\!\!\perp_{\mathbb{U}-}}$ selects every co-term that runs with the term $()$. This would include co-terms like $\tilde{\mu}[()\,.c]$ and $\tilde{\mu}\_.c$ for commands c that are in $\perp\!\!\!\perp$, because they both reduce to the safe state c in one step. However, $\{()\}^{\perp\!\!\!\perp_{\mathbb{U}-}}$ would not include co-terms like $1 \cdot []$ or $\tilde{\mu}[\iota_1\,(x)\,.c \mid \iota_2\,(y)\,.c']$ since the commands $\langle () \| 1 \cdot [] \rangle$ and $\langle () \| \tilde{\mu}[\iota_1\,(x)\,.c \mid \iota_2\,(y)\,.c'] \rangle$ are stuck on an irrecoverable type error which is excluded from $\perp\!\!\!\perp$. As another example, $\{\}^{\perp\!\!\!\perp_{\mathbb{U}-}}$ would instead select *every* co-term, since the condition $\forall v \in \{\}, \langle v \| e \rangle \in \perp\!\!\!\perp$ is vacuously true for any e. Note that this fact about $\{\}^{\perp}$ (or $\{\}^{\perp\!\!\!\perp_{\mathbb{U}+}}$) holds regardless of the definition of $\perp\!\!\!\perp$, so that the $\perp\!\!\!\perp$-orthogonal complement of the empty space $(\emptyset, \emptyset)$ inside $\mathbb{U}$ always gives back all of $\mathbb{U}$, for any $\perp\!\!\!\perp$ and $\mathbb{U}$. *End example* 7.1.

While we often have a particular purpose in mind (like the above example of type-safe execution), we can temporarily ignore the particular details and just leave the safety $\perp\!\!\!\perp$ abstract for the time being. As we will see, the nature of orthogonality itself already gives us some interesting structure independent of our choices, without knowing anything about the particularities of terms and co-terms.

Orthogonality and intuitionistic negation

As an operation on interaction spaces, orthogonality has some inherently negating behavior: it selects a collection of positive elements (terms) with respect to a collection of negative elements (co-terms), and vice versa. We will see that this simple intuition reveals a fundamental connection between the orthogonality of interaction spaces and the negation connective in intuitionistic logic. As it turns out, basic properties of intuitionistic negation, both from a logical and computational perspective, are shared with the orthogonality operation. Furthermore, classical but non-intuitionistic properties of negation are invalid for orthogonality.

Recall from Chapter II that in the intuitionistic logic of natural deduction, negation can be encoded in terms of implication and falsehood: $\neg A = A \to \bot$. This encoding of negation is summarized by the following two derivations for $\neg$ introduction and elimination that are derived from the rules for $\supset$ and $\bot$:

$$\dfrac{\begin{matrix}\overline{A}^{\,x}\\ \vdots \\ \bot\end{matrix}}{\neg A}\,\neg I_x \qquad\qquad \dfrac{\neg A \quad A}{\bot}\,\neg E$$

Using the above derived rules for negation, we can give some schematic proofs involving negation and implication that hold in intuitionistic logic. For example, we have the *contrapositive* of an implication, $(A \supset B) \supset (\neg B \supset \neg A)$,

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\neg B}^{\;k} \qquad \dfrac{\overline{A \supset B}^{\;f} \quad \overline{A}^{\;x}}{B}\supset E}{\bot}\neg E}{\neg A}\neg I_x}{(\neg B) \supset (\neg A)}\supset I_k}{(A \supset B) \supset ((\neg B) \supset (\neg A))}\supset I_f$$

double negation introduction, $A \supset (\neg\neg A)$,

$$\dfrac{\dfrac{\dfrac{\overline{\neg A}^{\;k} \quad \overline{A}^{\;x}}{\bot}\neg E}{\neg\neg A}\neg I_k}{A \supset (\neg\neg A)}\supset I_x$$

and *triple negation elimination*, $(\neg\neg\neg A) \supset (\neg A)$,

$$\dfrac{\dfrac{\dfrac{\overline{\neg\neg\neg A}^{\;w} \qquad \dfrac{\dfrac{\overline{\neg A}^{\;k} \quad \overline{A}^{\;x}}{\bot}\neg E}{\neg\neg A}\neg I_k}{\bot}\neg E}{\neg A}\neg I_x}{(\neg\neg\neg A) \supset (\neg A)}\supset I_h$$

Furthermore, each of these proofs can also be written as a corresponding term in the simply-typed λ-calculus as follows:

$$\begin{aligned}
&\mathit{Contra} : (A \to B) \to (\neg B \to \neg A)\\
&\mathit{Contra} = \lambda f : A \to B.\lambda k : \neg B.\lambda x : A.k\ (f\ x)
\end{aligned}$$

$$\begin{aligned}
&\mathit{DNI} &&: A \to \neg\neg A\\
&\mathit{DNI} &&= \lambda x : A.\lambda k : \neg A.k\ x
\end{aligned}$$

$$\begin{aligned}
&\mathit{TNE} &&: \neg\neg\neg A \to \neg A\\
&\mathit{TNE} &&= \lambda h : \neg\neg\neg A.\lambda x : A.h\ (\lambda k : \neg A.k\ x)
\end{aligned}$$

Remark 7.1. The three terms *Contra*, *DNI*, and *TNE* have an important status for pure functional programming in languages like Haskell. In particular, they give us a definition of the *continuation monad* over the return type $\bot$, $\mathit{Cont}\ A = \neg\neg A$.

Double negation introduction, *DNI*, is the *return* (*a.k.a. unit*) function. Triple negation elimination, *TNE*, is the *join* function from *Cont* (*Cont* *A*) → *Cont* *A* with a more general type. And *Contra* is the contravariant mapping function for the underlying ¬ functor. We can get the *Functor* mapping function *fmap* by *Contra*-mapping a function twice, $\mathit{fmap}\ f = \mathit{Contra}\ (\mathit{Contra}\ f)$. *End remark* 7.1.

As it turns out, these three properties of contrapositive mapping, double negation introduction, and triple negation elimination correspond to similar properties of orthogonality. In particular, the orthogonal complement of an interaction space takes on the role of negation, and the containment relation takes on the role of implication. With this correspondence in mind, we get the following three well-known intuitionistic orthogonality properties:

Property 7.1 (Intuitionistic orthogonality)**.** For any two poles $\mathbb{Q} \subseteq \mathbb{P}$ and $\mathbb{P}$-spaces $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$,

a) *contrapositive:* $\mathbb{A} \sqsubseteq \mathbb{B}$ implies $\mathbb{B}^{\mathbb{Q}\mathbb{C}} \sqsubseteq \mathbb{A}^{\mathbb{Q}\mathbb{C}}$,

b) *double orthogonal introduction:* $\mathbb{A} \sqsubseteq \mathbb{C}$ implies $\mathbb{A} \sqsubseteq \mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}}$, and

c) *triple orthogonal elimination:* $\mathbb{A} \sqsubseteq \mathbb{C}$ implies $\mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}} = \mathbb{A}^{\mathbb{Q}\mathbb{C}}$.

Proof. a) Suppose that $v \in \mathbb{B}^{\mathbb{Q}\mathbb{C}}$, so that by the definition of orthogonality, we know that $v \in \mathbb{C}$ and $\langle v \| e \rangle \in \mathbb{Q}$ for all $\in \mathbb{B}$. But since $\mathbb{A}$ is contained in $\mathbb{B}$, it follows that $\langle v \| e \rangle \in \mathbb{Q}$ for all $\in \mathbb{A}$, meaning that $v \in \mathbb{A}^{\mathbb{Q}\mathbb{C}}$ as well. Dually, $e \in \mathbb{B}^{\mathbb{Q}\mathbb{C}}$ implies that $e \in \mathbb{A}^{\mathbb{Q}\mathbb{C}}$ by the definition of orthogonality and the fact that $\mathbb{A}$ is contained in $\mathbb{B}$. Therefore, $\mathbb{B}^{\mathbb{Q}\mathbb{C}} \sqsubseteq \mathbb{A}^{\mathbb{Q}\mathbb{C}}$ follows from $\mathbb{A} \sqsubseteq \mathbb{B}$.

b) Suppose that $v \in \mathbb{A}$ and $e \in \mathbb{A}^{\mathbb{Q}\mathbb{C}}$. Since $\mathbb{A} \sqsubseteq \mathbb{C}$ it must be that $v \in \mathbb{C}$, and by the definition of orthogonality, it must also be that $\langle v \| e \rangle \in \mathbb{Q}$. But this also means that $v \in \mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}}$ by the definition of orthogonality as well. Dually, given any $e \in \mathbb{A}$, we also have that $e \in \mathbb{C}$ and $\langle v \| e \rangle \in \mathbb{Q}$ for all $v \in \mathbb{A}^{\mathbb{Q}\mathbb{C}}$, meaning that $e \in \mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}}$ as well. Therefore, $\mathbb{A} \sqsubseteq \mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}}$ follows from $\mathbb{A} \sqsubseteq \mathbb{C}$.

c) First, we get the fact that $\mathbb{A}^{\mathbb{Q}\mathbb{C}} \sqsubseteq \mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}}$ as an immediate consequence of double orthogonal introduction (Property 7.1 (b)) because $\mathbb{A}^{\mathbb{Q}\mathbb{C}} \sqsubseteq \mathbb{C}$ by definition of orthogonality. Second, we get $\mathbb{A} \sqsubseteq \mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}}$ from double orthogonal introduction (Property 7.1 (b)) again, from which $\mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}} \sqsubseteq \mathbb{A}^{\mathbb{Q}\mathbb{C}}$ follows by contrapositive (Property 7.1 (a)). Therefore, $\mathbb{A}^{\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}\mathbb{Q}\mathbb{C}} = \mathbb{A}^{\mathbb{Q}\mathbb{C}}$ follows from $\mathbb{A} \sqsubseteq \mathbb{C}$. □

It is important to point out that when demonstrating the above three properties, we never needed to know anything specific about the makeup of the computational poles $\mathbb{Q}$ and $\mathbb{P}$ or the interaction spaces $\mathbb{A}$, $\mathbb{B}$, or $\mathbb{B}$. No matter what choices me make, we get to use these intuitionistic reasoning principles when working with orthogonality. These are well-known properties of orthogonality (also noted by Munch-Maccagnoni (2009), for example).

Example 7.2. Recall from Remark 3.5 that one difference between negation in intuitionistic logic versus classical logic is that *double negation elimination*, i.e. $(\neg\neg A) \to A$, is not assumed to hold generically for any A in the intuitionistic setting. To see why "double orthogonal elimination", i.e. $\mathbb{A}^{\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}} \sqsubseteq \mathbb{A}$, does not hold in general, let's return to our example of type-safe execution from Example 7.1. For the moment, let's assume a call-by-value $\mathcal{V}$ evaluation strategy, so that every co-term is a co-value and thus $\langle \mu\alpha.c \| e\rangle \mapsto_{\mu_\mathcal{V}} c\{e/\alpha\}$ for any e. Recall that the orthogonal of the empty interaction space, $(\emptyset,\emptyset)^{\perp\!\!\!\perp_\mathbb{U}}$, is $\mathbb{U}$. Now, suppose that the command c is in the type-safe pole $\perp\!\!\!\perp$. Notice that $\langle \mu\_.c \| e\rangle \mapsto c$, so that for an arbitrary co-term e, the command $\langle \mu\_.c \| e\rangle$ reduces in one step to a command in $\perp\!\!\!\perp$. This means that the term $\mu\_.c$ must be in the double-orthogonal of the empty space, $\mu\_.c \in (\emptyset,\emptyset)^{\perp\!\!\!\perp_\mathbb{U}\perp\!\!\!\perp_\mathbb{U}}$. But this also means that we've run into a situation where the double orthogonal of an interaction space (namely the empty one) includes elements that weren't originally there. Therefore, in general we can't say that the double orthogonal gives back the same space that we started with.

Since taking the double orthogonal of a set of an interaction space can introduce new elements, we can view it as a closure operation. Furthermore, since taking the orthogonal thrice gives the same thing as just once (Property 7.1 (c)), flipping back and forth more than twice in this way is redundant: $\mathbb{A}^{\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}} = \mathbb{A}^{\mathbb{Q}_\mathbb{C}}$ and $\mathbb{A}^{\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}} = \mathbb{A}^{\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}}$, so only $\mathbb{A}$, $\mathbb{A}^{\mathbb{Q}_\mathbb{C}}$ and $\mathbb{A}^{\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}}$ are interesting. In this regard, $\mathbb{A}^{\mathbb{Q}_\mathbb{C}\mathbb{Q}_\mathbb{C}}$ can be seen as the completion of $\mathbb{A}$ with respect to the possible candidates in $\mathbb{C}$ and the criteria imposed by the pole $\mathbb{Q}$. *End example* 7.2.

By adding more connectives into the mix, like conjunction ($\wedge$) and disjunction ($\vee$) from Chapter II, we get additional properties of intuitionistic negation. In particular, we have the de Morgan law—used as the backbone of logical duality in Chapter III—that allows us to distribute negation over conjunction in both directions: $(\neg(A \vee B)) \leftrightarrow ((\neg A) \wedge (\neg B))$. This law is provable with the rules of NJ natural deduction as two implications:

$$\dfrac{\dfrac{\dfrac{\overline{\neg(A\lor B)}^{\,k} \quad \dfrac{\overline{A}^{\,x}}{A\lor B}\lor I_1}{\bot}\neg E}{\neg A}\neg I^x \qquad \dfrac{\dfrac{\overline{\neg(A\lor B)}^{\,k} \quad \dfrac{\overline{B}^{\,y}}{A\lor B}\lor I_2}{\bot}\neg E}{\neg B}\neg I_y}{\dfrac{(\neg A)\land(\neg B)}{(\neg(A\lor B))\supset((\neg A)\land(\neg B))}\supset I_k}\land I$$

$$\dfrac{\dfrac{\overline{A\lor B}^{\,x} \quad \dfrac{\dfrac{\dfrac{\overline{(\neg A)\land(\neg B)}^{\,k}}{\neg A}\land E_1 \quad \overline{A}^{\,y}}{\bot}\neg E}{\bot} \quad \dfrac{\dfrac{\dfrac{\overline{(\neg A)\land(\neg B)}^{\,k}}{\neg B}\land E_2 \quad \overline{B}^{\,z}}{\bot}\neg E}{\bot}}{\dfrac{\bot}{\bot}\neg I_x}\lor E_{y,z}}{((\neg A)\land(\neg B))\supset(\neg(A\lor B))}\supset I_k$$

We can also write down the terms corresponding to the above proofs in the simply-typed λ-calculus from Section 2.2, expressing the above de Morgan law as two functions:

$$\begin{aligned}
&\mathit{PairNeg} : ((\neg A)\times(\neg B)) \to (\neg(A+B))\\
&\mathit{PairNeg} = \lambda k.\lambda x.\,\mathbf{case}\,x\,\mathbf{of}\,\iota_1\,(y) \Rightarrow \pi_1(k)\ y \mid \iota_2\,(z) \Rightarrow \pi_2(k)\ z
\end{aligned}$$

$$\begin{aligned}
&\mathit{NegSum} : (\neg(A+B)) \to ((\neg A)\times(\neg B))\\
&\mathit{NegSum} = \lambda k.((\lambda x.k\ (\iota_1\,(x))),(\lambda y.k\ (\iota_2\,(y))))
\end{aligned}$$

There is another de Morgan law used for logical duality in Section 3.1 for distributing a negation over a conjunction in both directions: $(\neg(A\land B)) \leftrightarrow ((\neg A)\lor(\neg B))$. However, in an intuitionistic setting, this law does *not* hold both ways. In particular, we can only assume that the right-to-left direction of this law holds in general: $(\neg(A\land B)) \leftarrow ((\neg A)\lor(\neg B))$. This implication is provable in intuitionistic natural deduction:

$$\dfrac{\dfrac{\dfrac{}{(\neg A)\vee(\neg B)}\,k \qquad \dfrac{\dfrac{}{\neg A}\,q \quad \dfrac{\dfrac{}{A\wedge B}\,x}{A}\,\wedge E_1}{\bot}\,\neg E \qquad \begin{array}{c}\dfrac{\dfrac{}{\neg B}\,r \quad \dfrac{\dfrac{}{A\wedge B}\,x}{B}\,\wedge E_2}{\bot}\,\neg E\\ \bot\end{array}}{\dfrac{\bot}{\neg(A\wedge B)}\,\supset I_x}\,\vee E_{q,r}}{((\neg A)\vee(\neg B))\supset(\neg(A\wedge B))}\,\supset I_k$$

And we also have simplty-typed λ-calculus function that corresponds to the one direction of the law.

$$\begin{aligned}&\mathit{SumNeg} : ((\neg A)+(\neg B))\to(\neg(A\times B))\\&\mathit{SumNeg} = \lambda k.\lambda x.\,\mathbf{case}\,k\,\mathbf{of}\,\iota_1\,(q)\Rightarrow q\ (\pi_1(x))\mid\iota_2\,(r)\Rightarrow r\ (\pi_2(x))\end{aligned}$$

We are unable to write the inverse function, $\mathit{NegPair} : (\neg(A\times B))\to((\neg A)+(\neg B))$, since we don't know up front which of $\neg A$ or $\neg B$ to return in general.

Just like before, these three de Morgan laws correspond to similar properties of orthogonality. The following union and intersection operations on interaction spaces take on the roles of conjunction and disjunction, and they enjoy similar introduction and elimination properties as in the natural deduction logic of NJ.

Definition 7.5 (Union and intersection)**.** Given two $\mathbb{P}$-spaces $\mathbb{A} = (\mathbb{A}_+,\mathbb{A}_-)$ and $\mathbb{B} = (\mathbb{B}_+,\mathbb{B}_-)$, the *union of* $\mathbb{A}$ *and* $\mathbb{B}$, written $\mathbb{A}\sqcup\mathbb{B}$, and the *intersection of* $\mathbb{A}$ *and* $\mathbb{B}$, written $\mathbb{A}\sqcap\mathbb{B}$ is defined as:

$$\begin{aligned}(\mathbb{A}_+,\mathbb{A}_-)\sqcup(\mathbb{B}_+,\mathbb{B}_-) &\triangleq (\mathbb{A}_+\cup\mathbb{B}_+,\mathbb{A}_-\cup\mathbb{B}_-)\\(\mathbb{A}_+,\mathbb{A}_-)\sqcap(\mathbb{B}_+,\mathbb{B}_-) &\triangleq (\mathbb{A}_+\cap\mathbb{B}_+,\mathbb{A}_-\cap\mathbb{B}_-)\end{aligned}$$

Property 7.2 (Union/intersection introduction/elimination)**.** For any $\mathbb{P}$-spaces $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$,

a) $\mathbb{A}\sqsubseteq\mathbb{A}\sqcup\mathbb{B}$ and $\mathbb{B}\sqsubseteq\mathbb{A}\sqcup\mathbb{B}$,

b) $\mathbb{A}\sqcup\mathbb{B}\sqsubseteq\mathbb{C}$ if and only if $\mathbb{A}\sqsubseteq\mathbb{C}$ and $\mathbb{B}\sqsubseteq\mathbb{C}$,

c) $\mathbb{A}\sqcap\mathbb{B}\sqsubseteq\mathbb{A}$ and $\mathbb{A}\sqcap\mathbb{B}\sqsubseteq\mathbb{B}$, and

d) $\mathbb{C}\sqsubseteq\mathbb{A}\sqcap\mathbb{B}$ if and only if $\mathbb{C}\sqsubseteq\mathbb{A}$ and $\mathbb{C}\sqsubseteq\mathbb{B}$.

Proof. Each property follows from the definition of $\sqcup$ and $\sqcap$ in terms of the underlying set union and intersection operations. $\square$

Furthermore, when coupled with orthogonality, union and intersection give the following intuitionistic de Morgan orthogonality properties.

Property 7.3 (Spacial de Morgan laws)**.** For any poles $\mathbb{Q} \subseteq \mathbb{P}$ and $\mathbb{P}$-spaces $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$,

a) $(\mathbb{A} \sqcup \mathbb{B})^{\mathbb{Q}_\mathbb{C}} = \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcap \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$, and

b) $(\mathbb{A} \sqcap \mathbb{B})^{\mathbb{Q}_\mathbb{C}} \sqsupseteq \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcup \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$.

Proof. a) First, we show that $(\mathbb{A} \sqcup \mathbb{B})^{\mathbb{Q}_\mathbb{C}} \sqsubseteq \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcap \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$. Suppose that $v \in (\mathbb{A} \sqcup \mathbb{B})^{\mathbb{Q}_\mathbb{C}}$, so that by definition $v \in \mathbb{C}$ and $\langle v \| e \rangle \in \mathbb{Q}$ for all $e \in \mathbb{A} \sqcup \mathbb{B}$. By the definition of $\sqcup$, it follows that $\langle v \| e \rangle \in \mathbb{Q}$ for all $e \in \mathbb{A}$ and $\langle v \| e \rangle \in \mathbb{Q}$ for all $e \in \mathbb{B}$ separately, so we have that both $v \in \mathbb{A}^{\mathbb{Q}_\mathbb{C}}$ and $v \in \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$. Thus, $v \in \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcap \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$. Dually, every $e \in (\mathbb{A} \sqcup \mathbb{B})^{\mathbb{Q}_\mathbb{C}}$ also leads to $e \in \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcap \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$ for similar reasons.

Second, we show that $(\mathbb{A} \sqcup \mathbb{B})^{\mathbb{Q}_\mathbb{C}} \sqsupseteq \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcap \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$. Suppose that $v \in \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcap \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$. By the definition of $\sqcap$, it follows that both $v \in \mathbb{A}^{\mathbb{Q}_\mathbb{C}}$ and $v \in \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$, meaning that $v \in \mathbb{C}$, $\langle v \| e \rangle \in \mathbb{Q}$ for all $e \in \mathbb{A}$, and $\langle v \| e \rangle \in \mathbb{Q}$ for all $e \in \mathbb{B}$. Thus, $\langle v \| e \rangle \in \mathbb{Q}$ for all $e \in \mathbb{A} \sqcup \mathbb{B}$, so $v \in (\mathbb{A} \sqcup \mathbb{B})^{\mathbb{Q}_\mathbb{C}}$. Dually, every $e \in \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcap \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$ also leads to $e \in (\mathbb{A} \sqcup \mathbb{B})^{\mathbb{Q}_\mathbb{C}}$ for similar reasons.

b) Suppose that $v \in \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcup \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$, so that by the definition of $\sqcup$ and orthogonality we know $v \in \mathbb{C}$ and either $\langle v \| e \rangle \in \mathbb{Q}$ for all $e \in \mathbb{A}$ or $\langle v \| e \rangle \in \mathbb{Q}$ for all $e \in \mathbb{B}$. For any $e \in \mathbb{A} \sqcap \mathbb{B}$, it follows that $e \in \mathbb{A}$ and $e \in \mathbb{B}$ as well, so it must be that $\langle v \| e \rangle \in \mathbb{Q}$. Thus, $v \in (\mathbb{A} \sqcap \mathbb{B})^{\mathbb{Q}_\mathbb{C}}$. Dually, every $e \in \mathbb{A}^{\mathbb{Q}_\mathbb{C}} \sqcup \mathbb{B}^{\mathbb{Q}_\mathbb{C}}$ also leads to $e \in (\mathbb{A} \sqcap \mathbb{B})^{\mathbb{Q}_\mathbb{C}}$ for similar reasons. $\square$

Again, take notice that the de Morgan properties of orthogonality don't depend on what particular elements inhabit the computational poles or interaction spaces. They are general laws that come out from the definition of orthogonality and the other basic operations and relations on interaction spaces.

Example 7.3. To see why the de Morgan Property 7.3 (b) does not go both ways like Property 7.3 (a), let's return again to type-safe execution from Example 7.1. Now, suppose we begin with two sets of terms, $\mathbb{A}_+ = \{True, ()\}$ and $\mathbb{B}_+ = \{False, ()\}$, so that their intersection is $\mathbb{A}_+ \cap \mathbb{B}_+ = \{()\}$. The negative $⫫$-orthogonal of this intersection in $\mathbb{U}$ is $(\mathbb{A}_+ \cap \mathbb{B}_+)^{⫫_{\mathbb{U}_-}} = \{()\}^{⫫_{\mathbb{U}_-}} = \{e \mid \langle () \| e \rangle \in ⫫\}$. By the definition of $⫫$ from Example 7.1, given a command c in $⫫$ we have that the co-term $\tilde{\mu}[()\,.c]$ runs with $()$ because $\langle () \| \tilde{\mu}[()\,.c] \rangle \mapsto c$, so $\tilde{\mu}[()\,.c]$ is in $(\mathbb{A}_+ \cap \mathbb{B}_+)^{⫫_{\mathbb{U}_-}}$. However, both $\langle \mathsf{True} \| \tilde{\mu}[()\,.c] \rangle$ and $\langle \mathsf{False} \| \tilde{\mu}[()\,.c] \rangle$ are not in $⫫$ since they are stuck on a type error. This means that the co-term $\tilde{\mu}[()\,.c]$ is in neither $\mathbb{A}_+^{⫫_{\mathbb{U}_-}}$ nor $\mathbb{A}_+^{⫫_{\mathbb{U}_-}}$, since $\tilde{\mu}[()\,.c]$ fails to be type safe when run with some of the terms in $\mathbb{A}_+$ and $\mathbb{B}_+$. Therefore, we've stumbled onto a situation where a co-term is in the space orthogonal to an intersection, but does not come from the union of the separate orthogonal spaces. In other words, taking the orthogonal of an intersection between two sets of terms permits *more* possible co-terms than just forming the orthogonal sets in isolation and putting them together. *End example* 7.3.

Computation, Worlds, and Types

With the basic building blocks of computational poles, interaction spaces, and orthogonality at hand, we can now set the stage for constructing models of programming languages using these concepts. In particular, we will be modeling some safety condition of the language represented by a pole $⫫$, which in the context of the sequent calculus will contain commands exhibiting the desired property. While we have a lot of leeway in choosing $⫫$, it cannot be arbitrary, however. Because the purpose of the programming language is to compute, a safety condition must respect computation. For this reason, the safety condition is made up of three poles:

- The "top" pole $⫪$, which is unsafe, and corresponds to everything that can be written,

- The "bottom" pole $⫫$, which is the safe subset of $⫪$, and corresponds to only those programs which pass our criteria, and

- The "middle" pole $\mathbb{I}$, which is partially safe and lies between $⫪$ and $⫫$.

The purpose of $\mathbb{I}$ is to act as a waypoint toward safety: the elements of $\mathbb{I}$ are not quite safe yet, however, we have the assurance that all the elements of $\mathbb{I}$ that step into $⫫$ are safe. This is commonly known as *closure under expansion*, and in our notation is written as: for all $c \in \mathbb{I}$, if $c \rightsquigarrow c' \in ⫫$ then $c \in ⫫$.

Definition 7.6 (Safety condition)**.** A *safety condition* $\mathbb{S}$ is a triple of poles $(\Top, \mathbb{I}, \Bot)$ such that $\Bot \subseteq \mathbb{I} \subseteq \Top$ and the following condition holds:

- *Closure under expansion:* for all $c \in \mathbb{I}$, if $c \rightsquigarrow c' \in \Bot$ then $c \in \Bot$.

We call $\Top$ the *unsafe pole*, $\mathbb{I}$ the *demisafe pole*, and $\Bot$ the *safe pole* of the safety condition.

The fact that we are allowed to choose a $\mathbb{I}$ which is smaller than $\Top$ for the purposes of constraining closure under expansion is important for some applications, but not others. For instance, in the following Section 7.5 we can just take $\mathbb{I} = \Top$ for the goal of proving logical consistency and type safety. However, to show strong normalization it is crucial that we constrain $\mathbb{I}$ to include only the commands which may not be strongly normalizing themselves, but are formed by cutting together strongly normalizing (co-)terms. This restriction on $\mathbb{I}$ gives us a key foothold for demonstrating the closure under expansion which would be impossible otherwise.

In order to model the semantic meaning of types in terms of a chosen safety condition, we must delineate the world in which they reside. A world containing semantic types is represented as an interaction space which holds every possible element of all the types we're interested in. Therefore, it may allow for undesired interactions by mixing elements that belong to different types, but each inhabitant of the world should act as a well-behaved member of some potential type. To phrase this requirement, worlds are made up of three interaction spaces that represent the impact of substitution strategy in the programming language:

- The "untyped" interaction space $\mathbb{U}$ corresponds to everything that can be written without any restrictions,
- The "value" interaction space $\mathbb{V}$ is contained within $\mathbb{U}$ and corresponds to the values and co-values of a substitution strategy, and
- The "well-behaved" interaction space $\mathbb{W}$ is contained within $\mathbb{U}$ and corresponds to the elements which pass the minimum criteria needed to be considered elegible for belonging to a type.

The definition of a world in this sense makes heavy use of *(co-)value restriction*, since in many places computation can only proceed with (co-)values and not general (co-)terms. For this purpose, we need to know what it means to restrict one interaction space by another.

Definition 7.7 (Restriction)**.** Given two $\mathbb{P}$-spaces $\mathbb{A}$ and $\mathbb{B}$, the $\mathbb{B}$-*restriction of* $\mathbb{A}$, written $\mathbb{A}|_{\mathbb{B}}$, is their intersection $\mathbb{A} \sqcap \mathbb{B}$. Likewise, we write $\mathbb{A}|_{\mathbb{B}} = \mathbb{A} \cap \mathbb{B}$ for the $\mathbb{B}$-*restriction of* $\mathbb{A}$ when $\mathbb{A}$ and $\mathbb{B}$ are sets.

Note that in the notation of restriction $\mathbb{A}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} = (\mathbb{A} \sqcap \mathbb{V})^{\perp\!\!\!\perp_{\mathbb{W}}}$, whereas $\mathbb{A}^{\perp\!\!\!\perp_{\mathbb{W}}}\big|_{\mathbb{V}} = \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{W}}} \sqcap \mathbb{V}$. The semantic notion of worlds can now be defined in terms of two criteria: *saturation* which forces a sufficient amount of elements from $\mathbb{U}$ into $\mathbb{W}$ by stating that the portion of $\mathbb{U}$ which steps to a safe command with all "benign" elements is well-behaved, and *generation* which states that any interaction space contained in $\mathbb{W}$ has only safe interaction if all the interactions with (co-)values are safe.

Definition 7.8 (Worlds)**.** Given a safety condition $\mathbb{S} = (\top\!\!\!\top, \mathbb{I}, \perp\!\!\!\perp)$, an $\mathbb{S}$-*world* is a triple $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$ where both $\mathbb{U}$ and $\mathbb{V}$ are $\top\!\!\!\top$-spaces and $\mathbb{W}$ is a $\mathbb{I}$-space such that $\mathbb{V} \sqsubseteq \mathbb{U}$, $\mathbb{W} \sqsubseteq \mathbb{U}$, and the following conditions hold:

- *Saturation:* for all $v \in \mathbb{U}$, if $\langle v \| E \rangle \rightsquigarrow c$ for some $c \in \perp\!\!\!\perp$ and all $E \in \mathbb{W}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\big|_{\mathbb{V}}$ then $v \in \mathbb{W}$. Dually, for all $e \in \mathbb{U}$, if $\langle V \| e \rangle \rightsquigarrow c$ for some $c \in \perp\!\!\!\perp$ and all $V \in \mathbb{W}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\big|_{\mathbb{V}}$ then $e \in \mathbb{W}$. In other words, $\mathbb{W}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\Big|_{\mathbb{V}}^{\perp\!\!\!\perp^{1}_{\mathbb{U}}} \sqsubseteq \mathbb{W}$ where $\perp\!\!\!\perp^1 = \{c \in \top\!\!\!\top \mid c \rightsquigarrow c' \in \perp\!\!\!\perp\}$.
- *Generation:* for all $\mathbb{I}$-spaces $\mathbb{A} \sqsubseteq \mathbb{W}$, if $\mathbb{A} = \mathbb{A}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}$ then $\mathbb{A} = \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{W}}}$.

We call $\mathbb{U}$ the *untyped* $\top\!\!\!\top$*-space*, $\mathbb{V}$ the *value* $\top\!\!\!\top$*-space*, and $\mathbb{W}$ the *well-behaved* $\mathbb{I}$*-space*. As shorthand, for any $\top\!\!\!\top$-space $\mathbb{A} \sqsubseteq \mathbb{U}$, we write $V \in \mathbb{A}$ to denote $V \in \mathbb{A}|_{\mathbb{V}}$ and $E \in \mathbb{A}$ to denote $E \in \mathbb{A}|_{\mathbb{V}}$.

Note that the generation property is a rephrasing of Munch-Maccagnoni's (2009) generation lemma, where we take it as an assumption instead of proving that it holds for a particular setting. In a particular world $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$, we can say that a semantic type is any space $\mathbb{A}$ contained in the well-behaved $\mathbb{W}$ where every positive element of $\mathbb{A}$ is safe when paired with every negative value element of $\mathbb{A}$, and vice versa.

Definition 7.9 (Semantic types)**.** Given a safety condition $\mathbb{S} = (\top\!\!\!\top, \mathbb{I}, \perp\!\!\!\perp)$ and $\mathbb{S}$-world $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$, a $\mathbb{T}$-*type* is any $\perp\!\!\!\perp$-space $\mathbb{A}$ such that $\mathbb{A} = \mathbb{A}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}$. We denote the set of all $\mathbb{T}$-types as *SemType*($\mathbb{T}$).

Note that by the definition of semantic types, each one must contain some minimum amount of "benign" elements that belong to every type that lives in its world. These benign elements are safe when paired with *any* well-behaved member of the world, and so they never cause any problems.

Lemma 7.1 (Type minimum). *For any $\mathbb{S}$-world $\mathbb{T} = (\Top, \mathbb{I}, \Bot)$ and $\mathbb{T}$-type $\mathbb{A}$, $\mathbb{W}^{\Bot_{\mathbb{W}}} \sqsubseteq \mathbb{W}|_{\mathbb{V}}^{\Bot_{\mathbb{W}}} \sqsubseteq \mathbb{A}$.*

Proof. First, note that $\mathbb{W}|_{\mathbb{V}} \sqsubseteq \mathbb{W}$. Also, because $\mathbb{A}$ is a $\mathbb{T}$-type, we know that $\mathbb{A} = \mathbb{A}|_{\mathbb{V}}^{\Bot_{\mathbb{W}}}$, so that by definition of orthogonality, $\mathbb{A} \sqsubseteq \mathbb{W}$ and thus $\mathbb{A}|_{\mathbb{V}} \sqsubseteq \mathbb{W}|_{\mathbb{V}} \sqsubseteq \mathbb{W}$. Finally, by contrapositive (Property 7.1 (a)) we get $\mathbb{W}^{\Bot_{\mathbb{W}}} \sqsubseteq \mathbb{W}|_{\mathbb{V}}^{\Bot_{\mathbb{W}}} \sqsubseteq \mathbb{A}|_{\mathbb{V}}^{\Bot_{\mathbb{W}}} = \mathbb{A}$. □

It sometimes happens that $\mathbb{W}^{\Bot_{\mathbb{W}}}$ is empty, and if that's the case then there is not necessarily anything in the positive or negative sides of a semantic type. However, in some applications, like strong normalization, we find that things like (co-)variables are benign, and can be safely assumed to inhabit every possible type. The existence of this minimum of every type lets us prove the type expansion property, which says that any "untyped" term which steps to a safe place for all co-values of a type *must* belong to that type, and vice versa.

Lemma 7.2 (Type expansion). *For any safety condition $\mathbb{S} = (\Top, \mathbb{I}, \Bot)$, $\mathbb{S}$-world $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$, and $\mathbb{T}$-type $\mathbb{A}$,*

1. *$v \in \mathbb{A}$ if $\langle v \| E \rangle \rightsquigarrow c \in \Bot$ for all $E \in \mathbb{A}|_{\mathbb{V}}$, and*

2. *$e \in \mathbb{A}$ if $\langle V \| e \rangle \rightsquigarrow c \in \Bot$ for all $V \in \mathbb{A}|_{\mathbb{V}}$.*

Proof.

1. Note that $\mathbb{W}|_{\mathbb{V}}^{\Bot_{\mathbb{W}}} \sqsubseteq \mathbb{A}$ by Lemma 7.1, and so $\mathbb{W}|_{\mathbb{V}}^{\Bot_{\mathbb{W}}}\Big|_{\mathbb{V}} \sqsubseteq \mathbb{A}|_{\mathbb{V}}$ by monotonicity (Property 7.4 (a)), so that $v \in \mathbb{W}$ by saturation of $\mathbb{T}$ because $\langle v \| E \rangle \rightsquigarrow \Bot$ for all $E \in \mathbb{W}|_{\mathbb{V}}^{\Bot_{\mathbb{W}}}\Big|_{\mathbb{V}} \sqsubseteq \mathbb{A}|_{\mathbb{V}}$. Thus, for all $E \in \mathbb{A}|_{\mathbb{V}}$, $\langle v \| E \rangle \in \mathbb{I}$ since $\mathbb{W}$ is a $\mathbb{I}$-space, and so $\langle v \| E \rangle \in \Bot$ by closure under expansion of $\mathbb{S}$. Therefore, $v \in \mathbb{A}|_{\mathbb{V}}^{\Bot_{\mathbb{W}}} = \mathbb{A}$ because $\mathbb{A}$ is a $\mathbb{T}$-type.

2. Analogous to part 1 by duality. □

Note the two steps of this proof, which forms a general procedure of justifying the presence of elements in a type. First, we must justify that we are dealing with something generally well-behaved that exists in the $\mathbb{I}$-space $\mathbb{W}$. Only then can we use closure under expansion of the safety condition to show that it is also safe with every (co-)value of the type.

The positive construction of types

We now consider two dual methods of constructing particular types inside of a world. The first is the *positive* method, which builds a type around a chosen set of values. In particular, given some world $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$, where $\mathbb{V} = (\mathbb{V}_+, \mathbb{V}_-)$ and $\mathbb{W} = (\mathbb{W}_+, \mathbb{W}_-)$, and a chosen set of well-behaved value elements $\mathbb{A}_{cons} \subseteq \mathbb{W}_+|_{\mathbb{V}_+}$ serving as the primitive *constructions*, we have the *positive* construction of the $\mathbb{T}$-type $Pos_{\mathbb{T}}(\mathbb{A}_{cons})$, defined as follows:

$$Pos_{\mathbb{T}}(\mathbb{A}_{cons}) \triangleq \left(\mathbb{A}_{cons}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)\Big|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}$$

To show that $Pos_{\mathbb{T}}(\mathbb{A}_{cons})$ is actually a $\mathbb{T}$-type, we need to demonstrate that $Pos_{\mathbb{T}}(\mathbb{A}_{cons}) = Pos_{\mathbb{T}}(\mathbb{A}_{cons})|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}$. To do so, we rely on some facts about restriction, and how they generalize the basic properties of the orthogonality operation (Property 7.1).

Property 7.4. For all $\mathbb{P}$-spaces $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$,

a) *restriction monotonicity:* $\mathbb{A} \sqsubseteq \mathbb{B}$ implies $\mathbb{A}|_{\mathbb{C}} \sqsubseteq \mathbb{B}|_{\mathbb{C}}$,

b) *restriction containment:* $\mathbb{A}|_{\mathbb{C}} \sqsubseteq \mathbb{A}$, and

c) *restriction idempotency:* $\mathbb{A}|_{\mathbb{C}}|_{\mathbb{C}} = \mathbb{A}|_{\mathbb{C}}$.

Proof. Each property follows from the definition of restriction in terms of intersection, and the introduction and elimination facts in Property 7.2. In particular, $\mathbb{A} \sqsubset \mathbb{B}$ implies $\mathbb{A}|_{\mathbb{C}} = \mathbb{A} \sqcap \mathbb{C} \sqsubseteq \mathbb{B} \sqcap \mathbb{C} = \mathbb{A}|_{\mathbb{C}}$, $\mathbb{A}|_{\mathbb{C}} = \mathbb{A} \sqcap \mathbb{C} \sqsubseteq \mathbb{A}$, and $\mathbb{A}|_{\mathbb{C}}|_{\mathbb{C}} = \mathbb{A} \sqcap \mathbb{C} \sqcap \mathbb{C} \sqsubseteq \mathbb{A} \sqcap \mathbb{C} = \mathbb{A}|_{\mathbb{C}}$. □

Property 7.5. For any two poles $\subseteq^O \mathbb{P}$ and $\mathbb{P}$-spaces $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{D}$

a) *restricted orthogonal:* $\mathbb{A}^{\mathbb{O}_{\mathbb{C}}}\Big|_{\mathbb{D}} = \mathbb{A}^{\mathbb{O}_{\mathbb{C}|_{\mathbb{D}}}}$,

b) *restricted contrapositive:* $\mathbb{A} \sqsubseteq \mathbb{B}$ implies $\mathbb{B}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\Big|_{\mathbb{D}} \sqsubseteq \mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\Big|_{\mathbb{D}}$,

c) *restricted double orthogonal introduction:* $\mathbb{A} \sqsubseteq \mathbb{C}$ implies $\mathbb{A}|_{\mathbb{D}} \sqsubseteq \mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\Big|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\Bigg|_{\mathbb{D}}$, and

d) *restricted triple orthogonal elimination:* $\mathbb{A} \sqsubseteq \mathbb{C}$ implies $\mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\Big|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\Bigg|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\Bigg|_{\mathbb{D}} = \mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\Big|_{\mathbb{D}}$.

Proof. The restricted orthogonal Property 7.5 (a) follows from the definitions of the restriction and orthogonality operations on interaction spaces. In particular, supposing $\mathbb{A} = (\mathbb{A}_+, \mathbb{A}_-)$, $\mathbb{C} = (\mathbb{C}_+, \mathbb{C}_-)$ and $\mathbb{D} = (\mathbb{D}_+, \mathbb{D}_-)$, we have

$$\begin{aligned}
\left.\mathbb{A}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}} &= \mathbb{A}^{\mathbb{O}_{\mathbb{C}}} \sqcap \mathbb{D} = (\mathbb{A}_+, \mathbb{A}_-)^{\mathbb{O}_{(\mathbb{C}_+,\mathbb{C}_-)}} \sqcap (\mathbb{D}_+, \mathbb{D}_-) \\
&= \left(\mathbb{A}_-^{\mathbb{O}_{\mathbb{C}_+}}, \mathbb{A}_+^{\mathbb{O}_{\mathbb{C}_-}}\right) \sqcap (\mathbb{D}_+, \mathbb{D}_-) = \left(\mathbb{A}_-^{\mathbb{O}_{\mathbb{C}_+}} \cap \mathbb{D}_+, \mathbb{A}_+^{\mathbb{O}_{\mathbb{C}_-}} \cap \mathbb{D}_-\right) \\
&= (\{v \in \mathbb{C}_+ \mid \forall e \in \mathbb{A}_-\ \langle v \| e\rangle \in \mathbb{O}\} \cap \mathbb{D}_+, \{e \in \mathbb{C}_- \mid \forall v \in \mathbb{A}_+\ \langle v \| e\rangle \in \mathbb{O}\} \cap \mathbb{D}_-) \\
&= (\{v \in \mathbb{C}_+ \cap \mathbb{D}_+ \mid \forall e \in \mathbb{A}_-\ \langle v \| e\rangle \in \mathbb{O}\}, \{e \in \mathbb{C}_- \cap \mathbb{D}_- \mid \forall v \in \mathbb{A}_+\ \langle v \| e\rangle \in \mathbb{O}\}) \\
&= \left(\mathbb{A}_-^{\mathbb{O}_{\mathbb{C}_+ \cap \mathbb{D}_+}}, \mathbb{A}_+^{\mathbb{O}_{\mathbb{C}_- \cap \mathbb{D}_-}}\right) = (\mathbb{A}_-, \mathbb{A}_+)^{\mathbb{O}_{(\mathbb{C}_+ \cap \mathbb{D}_+, \mathbb{C}_- \cap \mathbb{D}_-)}} = \mathbb{A}^{\mathbb{O}_{\mathbb{C} \sqcap \mathbb{D}}} = \mathbb{A}^{\mathbb{O}_{\mathbb{C}|_{\mathbb{D}}}}
\end{aligned}$$

The other properties follow from Property 7.5 (a), the intuitionistic facts of orthogonality in Property 7.1, as well as the monotonicity of restriction (Property 7.4 (a)). For the restricted contrapositive, $\mathbb{A} \sqsubseteq \mathbb{B}$ implies $\mathbb{A}|_{\mathbb{D}} \sqsubseteq \mathbb{B}|_{\mathbb{D}}$ by the monotonicity Property 7.4 (a) which implies $\mathbb{B}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}} \sqsubseteq \mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}$ by the contrapositive Property 7.1 (a) which implies $\left.\mathbb{B}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}} \sqsubseteq \left.\mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}}$ by the monotonicity of restriction again. For the restricted double orthogonal introduction, $A \sqsubseteq \mathbb{C}$ implies $\mathbb{A}|_{\mathbb{D}} \sqsubseteq \mathbb{C}|_{\mathbb{D}}$ by monotonicity (Property 7.4 (a)) which implies $\mathbb{A}|_{\mathbb{D}} \sqsubseteq \mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}|_{\mathbb{D}}}\mathbb{O}_{\mathbb{C}|_{\mathbb{D}}}} = \left.\left.\mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}}$ by ordinary double orthogonal introduction (Property 7.1 (b)) and Property 7.5 (a). For the restricted triple orthogonal elimination, again $A \sqsubseteq \mathbb{C}$ implies $\mathbb{A}|_{\mathbb{D}} \sqsubseteq \mathbb{C}|_{\mathbb{D}}$ by monotonicity (Property 7.4 (a)) which implies $\left.\left.\left.\mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}} = \mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}|_{\mathbb{D}}}\mathbb{O}_{\mathbb{C}|_{\mathbb{D}}}\mathbb{O}_{\mathbb{C}|_{\mathbb{D}}}} = \mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}|_{\mathbb{D}}}} = \left.\mathbb{A}|_{\mathbb{D}}^{\mathbb{O}_{\mathbb{C}}}\right|_{\mathbb{D}}$ by ordinary triple orthogonal elimination (Property 7.1 (c)) and Property 7.5 (a). □

Lemma 7.3 (Positive semantic types)**.** *For any safety condition $\mathbb{S}$, $\mathbb{W}$-world $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$, and $\mathbb{A}_{cons} \subseteq \mathbb{W}_+|_{\mathbb{V}_+}$, it must be that $Pos_{\mathbb{T}}(\mathbb{A}_{cons})$ is a $\mathbb{T}$-type.*

Proof.

$$\begin{aligned}
& Pos_{\mathbb{T}}(\mathbb{A}_{cons})|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} \\
={}& \left.\left.\left(\mathbb{A}_{cons}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)\right|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\right|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\right|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} && (\textit{Definition}) \\
={}& \left.\left.\left(\mathbb{A}_{cons}, \mathbb{A}_{cons}|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)\right|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\right|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\right|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} && (\textit{Property } 7.4\ (c))
\end{aligned}$$

$$= \left(\mathbb{A}_{cons}\big|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_-}^{\perp\!\!\!\perp_{\mathbb{W}_+}}\Bigg|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\Bigg|_{\mathbb{V}_-}^{\perp\!\!\!\perp_{\mathbb{W}_+}},\ \mathbb{A}_{cons}\big|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_-}^{\perp\!\!\!\perp_{\mathbb{W}_+}}\Bigg|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}} \right) \qquad (\textit{Definition})$$

$$= \left(\mathbb{A}_{cons}\big|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_-}^{\perp\!\!\!\perp_{\mathbb{W}_+}},\ \mathbb{A}_{cons}\big|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_-}^{\perp\!\!\!\perp_{\mathbb{W}_+}}\Bigg|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}} \right) \qquad (\textit{Property } 7.5\ (d))$$

$$= \left(\mathbb{A}_{cons}, \mathbb{A}_{cons}\big|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)\Big|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} \qquad (\textit{Definition})$$

$$= \left(\mathbb{A}_{cons}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)\Big|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} = \mathit{Pos}_{\mathbb{T}}(\mathbb{A}_{cons}) \qquad (\textit{Property } 7.4\ (c)) \quad \square$$

The *Pos* construction of types, which involves three applications of orthogonality interspersed with value restrictions, is more complex than the traditional bi-orthogonal construction of types which needs only two applications of orthogonality. This is because we do not assume anything about the chosen substitution strategy, so that there may be both non-values and non-(co-)values making the value restriction necessary at every step and inducing an extra application of orthogonality to ensure that the restricted version triple orthogonal elimination principle (used for showing that $\mathit{Pos}(\mathbb{A}_{cons})$ is indeed a semantic type) applies. However, if we assume that the negative side of $\mathbb{V}$ is universal (corresponding to the call-by-value $\mathcal{V}$ substitution strategy where all co-terms are co-values), then we can greatly simplify the positive construction of types to be the more traditional bi-orthogonal definition.

Lemma 7.4 (Positive bi-orthogonality)**.** *For any safety condition* $\mathbb{S}$*,* $\mathbb{W}$*-world* $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$*, and* $\mathbb{A}_{cons} \subseteq \mathbb{W}_+|_{\mathbb{V}_+}$*, if* $\mathbb{V}_- = \mathbb{U}_-$ *then* $\mathit{Pos}_{\mathbb{T}}(\mathbb{A}_{cons}) = (\mathbb{A}_{cons}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}})^{\perp\!\!\!\perp_{\mathbb{W}}} = (\mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}})$*.*

Proof. Note that because $\mathit{Pos}_{\mathbb{T}}(\mathbb{A}_{cons})$ must be a $\mathbb{T}$-type (Lemma 7.3), we know that $\mathit{Pos}_{\mathbb{T}}(\mathbb{A}_{cons}) = \mathit{Pos}_{\mathbb{T}}(\mathbb{A}_{cons})^{\perp\!\!\!\perp_{\mathbb{W}}}$ by generation of $\mathbb{T}$ (Definition 7.8), so we have the following equality:

$$\mathit{Pos}_{\mathbb{T}}(\mathbb{A}_{cons})$$

$$= \left(\mathbb{A}_{cons}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)\Big|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} \qquad (\textit{Definition})$$

$$= \left(\mathbb{A}_{cons}\big|_{\mathbb{V}_+}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_-}\right)^{\perp\!\!\!\perp_{\mathbb{W}}}\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} \qquad (\textit{Definition})$$

$$
\begin{aligned}
&= \left(\mathbb{A}_{cons}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_-}\right)^{\perp\!\!\!\perp_{\mathbb{W}}}\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} && (\mathbb{A}_{cons} \subseteq \mathbb{V}_+) \\
&= \left(\mathbb{A}_{cons}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)^{\perp\!\!\!\perp_{\mathbb{W}}}\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}} && (\mathbb{V}_- = \mathbb{U}_-) \\
&= \left(\mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_+}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_-}\right)^{\perp\!\!\!\perp_{\mathbb{W}}} && (\mathit{Definition}) \\
&= \left(\mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_+}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)^{\perp\!\!\!\perp_{\mathbb{W}}} && (\mathbb{V}_- = \mathbb{U}_-) \\
&= \left(\mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right) && (\mathit{Definition}) \\
&= \left(\mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)^{\perp\!\!\!\perp_{\mathbb{W}}} && (\mathit{Pos}_{\mathbb{T}}(\mathbb{A}_{cons}) = \mathit{Pos}_{\mathbb{T}}(\mathbb{A}_{cons})^{\perp\!\!\!\perp_{\mathbb{W}}}) \\
&= \left(\mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}\Big|_{\mathbb{V}_+}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}\right) && (\mathit{Definition}) \\
&= \left(\mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}\right) && (\mathit{Previous}) \\
&= \left(\mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_+}\perp\!\!\!\perp_{\mathbb{W}_-}}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right) && (\mathit{Property}\ 7.1\ (c)) \\
&= \left(\mathbb{A}_{cons}, \mathbb{A}_{cons}^{\perp\!\!\!\perp_{\mathbb{W}_-}}\right)^{\perp\!\!\!\perp_{\mathbb{W}}} && (\mathit{Definition}) \quad \square
\end{aligned}
$$

The negative construction of types

The dual to the positive method of constructing types is the *negative* method, which builds a type around a chosen set of co-values. In particular, given some world $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$, where $\mathbb{V} = (\mathbb{V}_+, \mathbb{V}_-)$ and $\mathbb{W} = (\mathbb{W}_+, \mathbb{W}_-)$, and a chosen set of well-behaved co-value elements $\mathbb{A}_{obs} \subseteq \mathbb{W}_-|_{\mathbb{V}_-}$ serving as the primitive *observations*, we have the *negative* construction of the $\mathbb{T}$-type $\mathit{Neg}_{\mathbb{T}}\mathbb{A}_{obs}$, defined as follows:

$$
\mathit{Neg}_{\mathbb{T}}(\mathbb{A}_{obs}) \triangleq \left(\mathbb{A}_{obs}^{\perp\!\!\!\perp_{\mathbb{W}_+}}, \mathbb{A}_{obs}\right)\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}\Bigg|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}
$$

Lemma 7.5 (Negative semantic types)**.** *For any safety condition* $\mathbb{S}$*,* $\mathbb{W}$*-world* $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$*, and* $\mathbb{A}_{obs} \subseteq \mathbb{W}_-|_{\mathbb{V}_-}$*, it must be that* $\mathit{Neg}_{\mathbb{T}}(\mathbb{A}_{obs})$ *is a* $\mathbb{T}$*-type.*

Proof.

$$
\mathit{Neg}_{\mathbb{T}}(\mathbb{A}_{obs})|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}
$$

$$
\begin{aligned}
&= \left(\mathbb{A}_{obs}^{\perp\!\!\!\perp_{\mathbb{W}_+}}, \mathbb{A}_{obs}\right)\Big|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}}\bigg|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}}\Bigg|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}} && (\textit{Definition}) \\
&= \left(\mathbb{A}_{obs}\big|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-}, \mathbb{A}_{obs}\right)\Big|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}}\bigg|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}}\Bigg|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}} && (\textit{Property } 7.4\ (c)) \\
&= \left(\mathbb{A}_{obs}\big|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-}\Big|^{\perp\!\!\!\perp_{\mathbb{W}_-}}_{\mathbb{V}_+}\bigg|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-},\ \mathbb{A}_{obs}\big|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-}\Big|^{\perp\!\!\!\perp_{\mathbb{W}_-}}_{\mathbb{V}_+}\bigg|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-}\Bigg|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_+}\right) && (\textit{Definition}) \\
&= \left(\mathbb{A}_{obs}\big|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-}\Big|^{\perp\!\!\!\perp_{\mathbb{W}_-}}_{\mathbb{V}_+}\bigg|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-},\ \mathbb{A}_{obs}\big|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-}\Big|^{\perp\!\!\!\perp_{\mathbb{W}_-}}_{\mathbb{V}_+}\right) && (\textit{Property } 7.5\ (d)) \\
&= \left(\mathbb{A}_{obs}\big|^{\perp\!\!\!\perp_{\mathbb{W}_+}}_{\mathbb{V}_-}, \mathbb{A}_{obs}\right)\Big|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}}\bigg|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}} && (\textit{Definition}) \\
&= \left(\mathbb{A}_{obs}^{\perp\!\!\!\perp_{\mathbb{W}_+}}, \mathbb{A}_{obs}\right)\Big|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}}\bigg|^{\perp\!\!\!\perp_{\mathbb{W}}}_{\mathbb{V}} = \mathit{Neg}_{\mathbb{T}}(\mathbb{A}_{obs}) && (\textit{Property } 7.4\ (c)) \quad \square
\end{aligned}
$$

Similar to the fact that the positive construction of types *Pos* can be simplified to the traditional bi-orthogonal construction under certain assumptions about $\mathbb{V}$, the same holds for the negative construction of types *Neg*. Unsurprisingly, this requires the dual assumption that the positive side of $\mathbb{V}$ is universal (corresponding to the call-by-name substitution strategy $\mathbb{N}$ where all terms are values).

Lemma 7.6 (Negative bi-orthogonality)**.** *For any safety condition* $\mathbb{S}$*,* $\mathbb{W}$*-world* $\mathbb{T} = (\mathbb{U}, \mathbb{V}, \mathbb{W})$*, and* $\mathbb{A}_{obs} \subseteq \mathbb{W}_-|_{\mathbb{V}_-}$*, if* $\mathbb{V}_+ = \mathbb{U}_+$ *then* $\mathit{Neg}_{\mathbb{T}}(\mathbb{A}_{obs}) = (\mathbb{A}_{obs}^{\perp\!\!\!\perp_{\mathbb{W}_+}}, \mathbb{A}_{obs})^{\perp\!\!\!\perp_{\mathbb{W}}} = (\mathbb{A}_{obs}^{\perp\!\!\!\perp_{\mathbb{W}_+}}, \mathbb{A}_{obs}^{\perp\!\!\!\perp_{\mathbb{W}_-}\perp\!\!\!\perp_{\mathbb{W}_+}})$*.*

Proof. Analogous to the proof of Lemma 7.4 by duality. □

Models

We now build a parameterized model for the $\mu\tilde{\mu}$-calculus in earnest by partially instantiating the notion of safety conditions and world. Since one of the applications of interest in Section 7.5 is the binary property of *contextual equivalence*, we will make a model out of pairs of commands and (co-)terms. More specifically, our model is parameterized by an arbitrary safety condition $\mathbb{S} = (\top\!\!\!\top, \amalg, \perp\!\!\!\perp)$ as well as a world

$\mathbb{T}_{\mathcal{S}} = (\mathbb{U}_{\mathcal{S}}, \mathbb{V}_{\mathcal{S}}, \mathbb{W}_{\mathcal{S}})$ for every base kind $\mathcal{S}$ such that

$$\begin{aligned}
\mathbb{T} &= \mathit{Command} \times \mathit{Command} \\
\mathbb{U}_{\mathcal{S}} &= (\mathit{Term}_{\mathcal{S}} \times \mathit{Term}_{\mathcal{S}}, \mathit{CoTerm}_{\mathcal{S}} \times \mathit{CoTerm}_{\mathcal{S}}) \\
\mathbb{V}_{\mathcal{S}} &= (\mathit{Value}_{\mathcal{S}} \times \mathit{Value}_{\mathcal{S}}, \mathit{CoValue}_{\mathcal{S}} \times \mathit{CoValue}_{\mathcal{S}})
\end{aligned}$$

from the well-kinded (but untyped) syntax of the $\mu\tilde{\mu}$-calculus, and where the computation relation $\rightsquigarrow$ for the top pole $\mathbb{T}$ is defined as

$$\begin{aligned}
&(c_1, c_2) \rightsquigarrow (c_1', c_2') \text{ if } (c_1 \mapsto c_1') \wedge (c_2 \mapsto c_2') \\
&(c_1, c_2) \rightsquigarrow (c_1', c_2) \text{ if } c_1 \mapsto c_1' \\
&(c_1, c_2) \rightsquigarrow (c_1, c_2') \text{ if } c_2 \mapsto c_2'
\end{aligned}$$

and the cut operation for each $\mathbb{T}$-space $\mathbb{U}_{\mathcal{S}}$ is defined as

$$\langle (v_1, v_2) \| (e_1, e_2) \rangle \triangleq (\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle)$$

Therefore, the definitions of $\mathbb{I}$, $\perp\!\!\!\perp$, and $\mathbb{W}_{\mathcal{S}}$ for each strategy $\mathbb{S}$ is arbitrary, so long as they satisfy the criteria imposed by the safety condition $(\mathbb{T}, \mathbb{I}, \perp\!\!\!\perp)$ and worlds $(\mathbb{U}_{\mathcal{S}}, \mathbb{V}_{\mathcal{S}}, \mathbb{W}_{\mathcal{S}})$. Since we are dealing with binary relations, not just unary predicates, we will use the following shorthand:

- $c \perp\!\!\!\perp c'$ means $(c, c') \in \perp\!\!\!\perp$ and $c \;\mathbb{I}\; c'$ means $(c, c') \in \mathbb{I}$,
- $v \;\mathbb{A}\; v'$ means $(v, v') \in \mathbb{A}$ for any $\mathbb{A} \sqsubseteq \mathbb{U}_{\mathcal{S}}$, and
- $e \;\mathbb{A}\; e'$ means $(e, e') \in \mathbb{A}$ for any $\mathbb{A} \sqsubseteq \mathbb{U}_{\mathcal{S}}$.

To accomodate the size indexes Ix and Ord, the model is also parameterized by a *size measurement*, defined as follows.

Definition 7.10. A *size measurement* is a set of ordinals $\mathbb{O}$ equipped with two constants $0, \infty \in \mathbb{O}$, a unary operation $+1 : \mathbb{O} \to \mathbb{O}$, and a well-founded (partial) order $<$ between elements of $\mathbb{O}$ such that the following conditions hold:

1. 0 is less than ∞: $0 < \infty$,
2. $+1$ is monotonic: $\mathbb{M} < \mathbb{N}$ implies $+1(\mathbb{M}) < +1(\mathbb{N})$ for all $\mathbb{M}, \mathbb{N} \in \mathbb{O}$,

3. $+1$ is strictly increasing: $M < +1(M)$ for all $M \in \mathbb{O}$, and

4. ∞ is a limit of $+1$: $M < \infty$ implies $+1(M) < \infty$ for all $M \in \mathbb{O}$.

Types and Kinds

First, we build a model for the kinds and sorts in the wholly static part of the higher-order $\mu\tilde{\mu}$-calculus with structural recursion. Since the language of kinds includes functions and size indexes in addition to base kinds, we need to form the *Universe* containing all the semantic representations of the different kinds of syntactic types. In particular, base kinds are interpreted as the set of semantic types of the corresponding world, the kind of size type indexes are interpreted as the set of ordinals, and the kinds of type functions are interpreted as the set of partial functions (denoted by $\rightharpoonup$) between other members of the universe.

Definition 7.11 (Universe)**.** The *Universe* is the smallest set such that

1. $\mathbb{O} \in \mathit{Universe}$,

2. $\mathit{SemType}(\mathbb{T}_{\mathcal{S}}) \in \mathit{Universe}$ for all base kinds,

3. $(\mathbb{K} \rightharpoonup \mathbb{L}) \in \mathit{Universe}$ for all $\mathbb{K}, \mathbb{L} \in \mathit{Universe}$,

4. $\mathbb{K} \in \mathit{Universe}$, for all $\mathbb{K} \subseteq \mathbb{L} \in \mathit{Universe}$.

For any $\mathbb{K}, \mathbb{L} \in \mathit{Universe}$, $\mathbb{A} \in \mathbb{K}$, and $\mathbb{B} \in \mathbb{L}$, the partial function application $\mathbb{A}(\mathbb{B})$ is defined whenever there exists $\mathbb{K}_1, \mathbb{K}_2 \in \mathit{Universe}$ such that $\mathbb{A} \in \mathbb{K}_1 \to \mathbb{K}_2$ and $\mathbb{B} \in \mathbb{K}_1$, and is undefined otherwise.

With the universe in place, we can now define the meaning of kinds as a relationship between the *syntax* of types and their *semantics* (Pitts, 1997) in the model.

Definition 7.12 (Semantic kinds)**.** A *semantic kind* $\mathbb{K} \in \mathit{SemKind}$ is a pair $(\mathbb{D}, \mathbb{R})$ where $\mathbb{D} \in \mathit{Universe}$ and $\mathbb{R} \subseteq (\mathit{Type} \times \mathbb{D})$. We refer to $\mathbb{D}$ as the *domain* of $\mathbb{K}$ and $\mathbb{R}$ as the *syntactic-semantic relationship* of $\mathbb{K}$. As shorthand, given $\mathbb{K} \in \mathit{SemKind}$, $A \in \mathit{Type}$, and $\mathbb{A} \in \mathit{Universe}$, we write $\mathbb{A} \in \mathbb{K}$ to indicate $\mathbb{A} \in \pi_1(\mathbb{K})$ and $A\ \mathbb{K}\ \mathbb{A}$ to indicate $(A, \mathbb{A}) \in \pi_2(\mathbb{K})$.

First, we give an interpretation of sorts in terms of semantic kinds. The sort of non-erasable kinds, ■, is interpreted as the whole set of all semantic kinds, and the sort of erasable kinds, □, adds the restriction that the syntactic-semantic relation is closed under β expansion of the syntactic component.

$$\llbracket \blacksquare \rrbracket \triangleq \mathit{SemKind} \qquad \llbracket \square \rrbracket \triangleq \{\mathbb{K} \in \mathit{SemKind} \mid \forall A' \; \mathbb{K} \; \mathbb{A}. \forall A \twoheadrightarrow_\beta A'. A \; \mathbb{K} \; \mathbb{A}\}$$

A *type substitution* σ is a partial function from syntactic type variables and connective names to semantic entities in some kind in the universe, and the set of all type substitutions is

$$\mathit{TypeSubstitution} \triangleq (\mathit{TypeVariable} \cup \mathit{Connective}) \rightharpoonup \bigcup \mathit{Universe}$$

The interpretation of syntactic kinds and types is then a (partial) function from type substitutions to semantic entities in *SemKind* and *Universe*, respectively.

$$\begin{aligned} \llbracket k \in \mathit{Kind} \rrbracket &: \mathit{TypeSubstitution} \rightharpoonup \mathit{SemKind} \\ \llbracket A \in \mathit{Type} \rrbracket &: \mathit{TypeSubstitution} \rightharpoonup \mathit{Universe} \end{aligned}$$

This interpretation is mutually defined by structural induction over the syntax.

$$\begin{aligned} \llbracket X \rrbracket_\sigma &\triangleq \sigma(X) \\ \llbracket 0 \rrbracket_\sigma &\triangleq 0 \\ \llbracket \infty \rrbracket_\sigma &\triangleq \infty \\ \llbracket N + 1 \rrbracket_\sigma &\triangleq +1(\llbracket N \rrbracket_\sigma) \\ \left\llbracket \mathsf{F}(\overrightarrow{A}) \right\rrbracket_\sigma &\triangleq \sigma(\mathsf{F})(\overrightarrow{\llbracket A \rrbracket_\sigma}) \\ \llbracket \lambda X{:}k.B \rrbracket_\sigma &\triangleq \lambda \mathbb{X} \in \pi_1(\llbracket k \rrbracket_\sigma).\llbracket B \rrbracket_{\sigma\{\mathbb{X}/X\}} \\ \llbracket A \; B \rrbracket_\sigma &\triangleq \llbracket A \rrbracket_\sigma(\llbracket B \rrbracket_\sigma) \end{aligned}$$

$$\begin{aligned} \llbracket \mathcal{S} \rrbracket_\sigma &\triangleq (\mathit{SemType}(\mathbb{T}_\mathcal{S}), \mathit{Type} \times \mathit{SemType}(\mathbb{T}_\mathcal{S})) \\ \llbracket k \to l \rrbracket_\sigma &\triangleq (\pi_1(\llbracket k \rrbracket_\sigma) \to \pi_1(\llbracket l \rrbracket_\sigma), \{(A, \mathbb{A}) \mid \forall B \; \llbracket k \rrbracket_\sigma \; \mathbb{B}. A \; B \; \llbracket l \rrbracket_\sigma \; \mathbb{A}(\mathbb{B})\}) \\ \llbracket \mathsf{Ix} \rrbracket_\sigma &\triangleq \left(\mathbb{N}, \left\{(M, \mathbb{M}) \mid M \sim^{\mathbb{N}} \mathbb{M}\right\}\right) \\ \llbracket \mathsf{Ord} \rrbracket_\sigma &\triangleq \left(\mathbb{O}, \left\{(M, \mathbb{M}) \mid \exists M' \in \mathit{Type}. M \twoheadrightarrow_\beta M' \sim^{\mathbb{O}} \mathbb{M}' \leq \mathbb{M}\right\}\right) \end{aligned}$$

$$\llbracket < N \rrbracket_\sigma \triangleq \left(\{ \mathbb{M} \in \mathbb{O} \mid \mathbb{M} < \llbracket N \rrbracket_\sigma \}, \left\{ (M, \mathbb{M}) \mid \exists M' \in \mathit{Type}. M \twoheadrightarrow_\beta M' \sim^{\mathbb{O}} \mathbb{M}' \leq \mathbb{M} \right\} \right)$$

Note that $\mathbb{N}$ is defined as the smallest subset of $\mathbb{O}$ containing 0 and closed under $+1$, the relation $\sim^{\mathbb{N}}$ is defined as the smallest subset of $\mathit{Type} \times \mathbb{O}$ such that

1. $0 \sim^{\mathbb{N}} 0$, and
2. $M + 1 \sim^{\mathbb{N}} +1(\mathbb{M})$ for all $M \sim^{\mathbb{N}} \mathbb{M}$.

and the relation $\sim^{\mathbb{O}}$ used in is defined as $\sim^{\mathbb{N}} \cup \{(\infty, \infty)\}$.

Declarations

Using the interpretation of kinds above, each (co-)data declaration can be interpreted as the semantic type representing the connective it defines.

$$\llbracket \mathit{decl} \in \mathit{Declaration} \rrbracket : \mathit{TypeSubstitution} \rightharpoonup \mathit{Universe}$$

The interpretations revolve around structures: data types are interpreted as the positive type built around their constructions, and co-data types are interpreted as the negative type built around their observations. We consider each different form of (co-)data type declaration introduced previously in Chapters V and VI, first warming up with simple (co-)data types before moving on to higher-order (co-)data types and recursive (co-)data types.

Simple (co-)data types

To interpret a data type, we must interpret the meaning of each of its constructors. In particular, given the signature of a constructor,

$$\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B : \mathcal{R}} \right)$$

in a data type declaration, we define its interpretation as the relation between the possible term constructions it can build, where the constructors agree and the sub-(co-)terms are related.

$$\left\llbracket \mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B : \mathcal{R}} \right) \right\rrbracket_\sigma \triangleq \left\{ \left(\mathsf{K}(\overrightarrow{e}, \overrightarrow{v}), \mathsf{K}(\overrightarrow{e'}, \overrightarrow{v'}) \right) \middle| \overrightarrow{e \llbracket B \rrbracket_\sigma e'}, \overrightarrow{v \llbracket A \rrbracket_\sigma v'} \right\}$$

The interpretation of a full simple data type declaration is then the function returning the positive type built around the union of all of its constructions as follows:

$$\left\llbracket \begin{array}{l} \mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where} \\ \qquad \overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^j \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^j\right)}^i \end{array} \right\rrbracket_\sigma$$

$$\triangleq \overrightarrow{\lambda \mathbb{X} \in \pi_1(\llbracket k \rrbracket_\sigma)}.Pos_{\mathbb{T}_\mathcal{S}} \left(\bigcup_i \left\{ \left\llbracket \mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^j \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^j\right) \right\rrbracket_{\sigma\left\{\overrightarrow{\mathbb{X}/X}\right\}} \right\} \right)$$

Co-data types are dual to data types, and follow the opposite approach. First we interpret the meaning of each of a co-data type's observers, given its signature $\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B : \mathcal{R}}\right)$, as the relation between the possible co-term observations it can build where the observers agree and the sub-(co-)terms are related.

$$\left\llbracket \mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B : \mathcal{R}}\right) \right\rrbracket_\sigma \triangleq \left\{ \left(\mathsf{O}[\vec{v}, \vec{e}], \mathsf{O}[\vec{v'}, \vec{e'}]\right) \middle| \overrightarrow{v \llbracket A \rrbracket_\sigma v'}, \overrightarrow{e \llbracket B \rrbracket_\sigma e'} \right\}$$

The interpretation of a full simple co-data type declaration is then the function returning the negative type built around the union of all of its observations as follows:

$$\left\llbracket \begin{array}{l} \mathbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where} \\ \qquad \overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^j \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^j\right)}^i \end{array} \right\rrbracket_\sigma$$

$$\triangleq \overrightarrow{\lambda \mathbb{X} \in \pi_1(\llbracket k \rrbracket_\sigma)}.Neg_{\mathbb{T}_\mathcal{S}} \left(\bigcup_i \left\{ \left\llbracket \mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^j \mid \mathsf{G}(\vec{X}) \vdash \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^j\right) \right\rrbracket_{\sigma\left\{\overrightarrow{\mathbb{X}/X}\right\}} \right\} \right)$$

Higher-order (co-)data types

The only difference between simple and higher-order (co-)data types is that higher-order (co-)data types can also include hidden quantified types within their structures. Therefore, when interpreting the meaning of their constructors and observers, we must also quantify over the possible types that might be included. For a quantified type of kind l, we extend the relation to quantify over $\lessdot[\sigma]$ twice, choosing a pair of syntactic types C and C' which are related to exactly the same semantic type $\mathbb{C}$. The two syntactic types are used in syntactic term and co-term structures, whereas the single semantic type is used for to interpret the types of the remaining components as follows:

$$\begin{aligned}&\left[\!\left[\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right)\right]\!\right]_{\sigma} \\ &\triangleq \left\{\left(\mathsf{K}^{\vec{C}}(\vec{e}, \vec{v}), \mathsf{K}^{\vec{C'}}(\vec{e'}, \vec{v'})\right) \middle| \overrightarrow{C \; [\![l]\!]_{\sigma} \; \mathbb{C}}, \overrightarrow{C' \; [\![l]\!]_{\sigma} \; \mathbb{C}}, \overrightarrow{e \; [\![B]\!]_{\sigma\left\{\overrightarrow{\mathbb{C}/Y}\right\}} \; e'}, \overrightarrow{v \; [\![A]\!]_{\sigma\left\{\overrightarrow{\mathbb{C}/Y}\right\}} \; v'}\right\}\end{aligned}$$

$$\begin{aligned}&\left[\!\left[\mathsf{O} : \left(\overrightarrow{A : \mathcal{T}} \mid \mathsf{G}(\vec{X}) \vdash^{\overrightarrow{Y:l}} \overrightarrow{B : \mathcal{R}}\right)\right]\!\right]_{\sigma} \\ &\triangleq \left\{\left(\mathsf{O}^{\vec{C}}[\vec{v}, \vec{e}], \mathsf{O}^{\vec{C'}}[\vec{v'}, \vec{e'}]\right) \middle| \overrightarrow{C \; [\![l]\!]_{\sigma} \; \mathbb{C}}, \overrightarrow{C' \; [\![l]\!]_{\sigma} \; \mathbb{C}}, \overrightarrow{v \; [\![A]\!]_{\sigma\left\{\overrightarrow{\mathbb{C}/Y}\right\}} \; v'}, \overrightarrow{e \; [\![B]\!]_{\sigma\left\{\overrightarrow{\mathbb{C}/Y}\right\}} \; e'}\right\}\end{aligned}$$

With this extended interpretation of single constructors and observers, the interpretation of higher-order (co-)data types is effectively the same as the simple case, defined as follows:

$$\begin{aligned}&\left[\!\!\left[\begin{array}{l}\textbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\textbf{where} \\ \qquad \overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y_{ij}:l_{ij}}^{j}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i}\end{array}\right]\!\!\right]_{\sigma} \\ &\triangleq \overrightarrow{\lambda\mathbb{X} \in \pi_1([\![k]\!]_{\sigma})}.Pos_{\mathbb{T}_{\mathcal{S}}}\left(\bigcup_i\left\{\left[\!\left[\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y_{ij}:l_{ij}}^{j}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)\right]\!\right]_{\sigma\left\{\overrightarrow{\mathbb{X}/X}\right\}}\right\}\right)\end{aligned}$$

$$\begin{aligned}&\left[\!\!\left[\begin{array}{l}\textbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\textbf{where} \\ \qquad \overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\vec{X}) \vdash^{\overrightarrow{Y_{ij}:l_{ij}}^{j}} \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i}\end{array}\right]\!\!\right]_{\sigma} \\ &\triangleq \overrightarrow{\lambda\mathbb{X} \in \pi_1([\![k]\!]_{\sigma})}.Neg_{\mathbb{T}_{\mathcal{S}}}\left(\bigcup_i\left\{\left[\!\left[\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\vec{X}) \vdash^{\overrightarrow{Y_{ij}:l_{ij}}^{j}} \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)\right]\!\right]_{\sigma\left\{\overrightarrow{\mathbb{X}/X}\right\}}\right\}\right)\end{aligned}$$

Primitive recursive (co-)data types

Interpreting recursively-defined (co-)data types is more interesting, since the interpretation itself must also be recursive. As it turns out, we can use the same recursion principle corresponding to the program to define the connective semantically.

In particular, we can interpret primitive-recursive data type as

$$\left[\!\!\left[\begin{array}{ll} \textbf{data}\,\mathsf{F}(i{:}\mathsf{Ix}, \overrightarrow{X{:}k}) : \mathcal{S}\,\textbf{by}\,\text{primitive recursion on } i & \\ \textbf{where}\, i = 0 & \overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_i{:}\mathcal{T}_i} \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{F}(0,\overrightarrow{X}) \mid \overrightarrow{B_i{:}\mathcal{R}_i}\right)}^{i} \\ \textbf{where}\, i = j{+}1 & \overrightarrow{\mathsf{K}'_i : \left(\overrightarrow{A'_i{:}\mathcal{T}'_i} \vdash^{\overrightarrow{Y'_i{:}l'_i}} \mathsf{F}(j{+}1,\overrightarrow{X}) \mid \overrightarrow{B'_i{:}\mathcal{R}'_i}\right)}^{i} \end{array}\right]\!\!\right]_{\sigma} \triangleq \lambda\mathbb{J}\in\mathbb{N}.\mathbb{F}^{\mathbb{J}}_{\sigma}$$

where the family of $\mathbb{F}^{\mathbb{M}}_{\sigma}$ is defined by primitive recursion on $\mathbb{M}\in\mathbb{N}$:

$$\begin{aligned} \mathbb{F}^{0}_{\sigma} &\triangleq \overrightarrow{\lambda\mathbb{X}\in\pi_1(\llbracket k\rrbracket).} \\ &\qquad Pos_{\mathbb{T}_{\mathcal{S}}}\left(\bigcup_{i}\left\{\left[\!\!\left[\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_i{:}\mathcal{T}_i} \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{F}(0,\overrightarrow{X}) \mid \overrightarrow{B_i{:}\mathcal{R}_i}\right)}^{i}\right]\!\!\right]_{\sigma\left\{\overrightarrow{\mathbb{X}/X}\right\}}\right\}\right) \\ \mathbb{F}^{+1(\mathbb{M})}_{\sigma} &\triangleq \overrightarrow{\lambda\mathbb{X}\in\pi_1(\llbracket k\rrbracket).} \\ &\qquad Pos_{\mathbb{T}_{\mathcal{S}}}\left(\bigcup_{i}\left\{\left[\!\!\left[\overrightarrow{\mathsf{K}'_i : \left(\overrightarrow{A'_i{:}\mathcal{T}'_i} \vdash^{\overrightarrow{Y'_i{:}l'_i}} \mathsf{F}(0,\overrightarrow{X}) \mid \overrightarrow{B'_i{:}\mathcal{R}'_i}\right)}^{i}\right]\!\!\right]_{\sigma\left\{\overrightarrow{\mathbb{X}/X},(\lambda\mathbb{J}\in\mathbb{N}.\mathbb{F}^{\mathbb{M}}_{\sigma})/F\right\}}\right\}\right) \end{aligned}$$

Note that this is well-defined by primitive recursion whenever the declaration of F is well-formed, because the signature of the constructors K'_i can only $\mathsf{F}(j,\overrightarrow{C})$, which is defined by the previous $\mathbb{F}$.

Dually, we can interpret a primitive-recursive co-data type as

$$\left[\!\!\left[\begin{array}{ll} \textbf{codata}\,\mathsf{G}(i{:}\mathsf{Ix}, \overrightarrow{X{:}k}) : \mathcal{S}\,\textbf{by}\,\text{primitive recursion on } i & \\ \textbf{where}\, i = 0 & \overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_i{:}\mathcal{T}_i} \mid \mathsf{G}(0,\overrightarrow{X}) \vdash^{\overrightarrow{Y_i{:}l_i}} \overrightarrow{B_i{:}\mathcal{R}_i}\right)}^{i} \\ \textbf{where}\, i = j{+}1 & \overrightarrow{\mathsf{O}'_i : \left(\overrightarrow{A'_i{:}\mathcal{T}'_i} \mid \mathsf{G}(j{+}1,\overrightarrow{X}) \vdash^{\overrightarrow{Y'_i{:}l'_i}} \overrightarrow{B'_i{:}\mathcal{R}'_i}\right)}^{i} \end{array}\right]\!\!\right]_{\sigma} \triangleq \lambda\mathbb{J}\in\mathbb{N}.\mathbb{G}^{\mathbb{J}}_{\sigma}$$

where the family of $\mathbb{G}^{\mathbb{M}}_{\sigma}$ is defined by primitive recursion on $\mathbb{M}\in\mathbb{N}$:

$$\begin{aligned} \mathbb{G}^{0}_{\sigma} &\triangleq \overrightarrow{\lambda\mathbb{X}\in\pi_1(\llbracket k\rrbracket).} \\ &\qquad Neg_{\mathbb{T}_{\mathcal{S}}}\left(\bigcup_{i}\left\{\left[\!\!\left[\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_i{:}\mathcal{T}_i} \mid \mathsf{G}(0,\overrightarrow{X}) \vdash^{\overrightarrow{Y_i{:}l_i}} \overrightarrow{B_i{:}\mathcal{R}_i}\right)}^{i}\right]\!\!\right]_{\sigma\left\{\overrightarrow{\mathbb{X}/X}\right\}}\right\}\right) \\ \mathbb{G}^{+1(\mathbb{M})}_{\sigma} &\triangleq \overrightarrow{\lambda\mathbb{X}\in\pi_1(\llbracket k\rrbracket).} \end{aligned}$$

$$Neg_{\mathbb{T}_{\mathcal{S}}}\left(\bigcup_i \left\{ \left[\!\!\left[\overrightarrow{\mathsf{O}'_i : \left(\overrightarrow{A'_i{:}\mathcal{T}'_i} \mid \mathsf{G}(0,\vec{X}) \vdash^{\overrightarrow{Y'_i{:}l'_i}} \overrightarrow{B'_i{:}\mathcal{R}'_i}\right)}^{i} \right]\!\!\right]_{\sigma\left\{\overrightarrow{\mathbb{X}/X},(\lambda\mathbb{J}\in\mathbb{N}.\mathbb{G}^{\mathbb{M}}_{\sigma})/G\right\}} \right\}\right)$$

<u>Noetherian recursive (co-)data types</u>

Data types defined by noetherian recursion are interpreted by the same recursion principle, so that we have the following interpretation

$$\left[\!\!\left[\begin{array}{ll} \textbf{data}\,\mathsf{F}(i:\mathsf{Ord},\overrightarrow{X:k}):\mathcal{S}\,\textbf{by}\text{ noetherian recursion on } i \\ \textbf{where} \qquad\qquad \mathsf{K}:\left(\overrightarrow{A:\mathcal{T}} \vdash^{\overrightarrow{Y:l}} \mathsf{F}(i,\vec{X}) \mid \overrightarrow{B:\mathcal{R}}\right) \end{array} \right]\!\!\right]_{\sigma} \triangleq \lambda\mathbb{J}\in\mathbb{O}.\mathbb{F}^{\mathbb{J}}_{\sigma}$$

where the family of $\mathbb{F}^{\mathbb{M}}_{\sigma}$ is defined by noetherian recursion on $\mathbb{M}\in\mathbb{O}$:

$$\begin{aligned} \mathbb{F}^{\mathbb{M}}_{\sigma} \triangleq{} & \overrightarrow{\lambda\mathbb{X}\in\pi_1([\![k]\!]).} \\ & Pos_{\mathbb{T}_{\mathcal{S}}}\left(\bigcup_i \left\{ \left[\!\!\left[\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_i{:}\mathcal{T}_i} \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{F}(i,\vec{X}) \mid \overrightarrow{B_i{:}\mathcal{R}_i}\right)}^{i} \right]\!\!\right]_{\sigma\left\{\overrightarrow{\mathbb{X}/X},\mathbb{M}/i,(\lambda\mathbb{J}\in(<\mathbb{M}).\mathbb{F}^{\mathbb{J}}_{\sigma})/\mathsf{F}\right\}} \right\}\right) \end{aligned}$$

where $<\mathbb{M}$ is the set $\{\mathbb{M}'\in\mathbb{O} \mid \mathbb{M}'<\mathbb{M}\}$. Note that this is well-defined by noetherian recursion whenever the declaration of F is well-formed, because each use of F in the signature of the constructors K_i must be on a strictly decreasing i (with respect to the $<$ ordering on ordinal indexes).

Dually, we can interpret a noetherian-recursive co-data type as

$$\left[\!\!\left[\begin{array}{ll} \textbf{codata}\,\mathsf{G}(i:\mathsf{Ord},\overrightarrow{X:k}):\mathcal{S}\,\textbf{by}\text{ noetherian recursion on } i \\ \textbf{where} \qquad\qquad \mathsf{O}:\left(\overrightarrow{A:\mathcal{T}} \mid \mathsf{G}(i,\vec{X}) \vdash^{\overrightarrow{Y:l}} \overrightarrow{B:\mathcal{R}}\right) \end{array} \right]\!\!\right]_{\sigma} \triangleq \lambda\mathbb{J}\in\mathbb{O}.\mathbb{G}^{\mathbb{J}}_{\sigma}$$

where the family of $\mathbb{G}^{\mathbb{M}}_{\sigma}$ is defined by noetherian recursion on $\mathbb{M}\in\mathbb{O}$:

$$\begin{aligned} \mathbb{G}^{\mathbb{M}}_{\sigma} \triangleq{} & \overrightarrow{\lambda\mathbb{X}\in\pi_1([\![k]\!]).} \\ & Neg_{\mathbb{T}_{\mathcal{S}}}\left(\bigcup_i \left\{ \left[\!\!\left[\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_i{:}\mathcal{T}_i} \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{G}(i,\vec{X}) \mid \overrightarrow{B_i{:}\mathcal{R}_i}\right)}^{i} \right]\!\!\right]_{\sigma\left\{\overrightarrow{\mathbb{X}/X},\mathbb{M}/i,(\lambda\mathbb{J}\in(<\mathbb{M}).\mathbb{G}^{\mathbb{J}}_{\sigma})/\mathsf{G}\right\}} \right\}\right) \end{aligned}$$

Sequents

The meaning of sequents is to constrain the potential domain of substitutions. That is, having $x : A$ in the input environment of a sequent means that we must substitute a suitable value for x which belongs to the interpretation of A. Since sequents quantify over several different types of variables (type variables, connectives, value variables and co-value variables), the substitution they correspond to must map each variable to an appropriate replacement as follows:

$$\begin{aligned}
\mathit{Substitution} \triangleq {} & ((\mathit{TypeVariable} \cup \mathit{Connective}) \rightharpoonup (\mathit{Type} \times \bigcup \mathit{Universe})) \\
& \times (\mathit{Variable} \rightharpoonup (\mathit{Value} \times \mathit{Value})) \\
& \times (\mathit{CoVariable} \rightharpoonup (\mathit{CoValue} \times \mathit{CoValue}))
\end{aligned}$$

Note that substitutions map (co-)variables to *pairs* of (co-)values, continuing the interpretation of types as binary relations on (co-)terms. These big substitutions can be pared down into more useful parts by extracting only the semantic component (via *sem*) used in the interpretation of types, as well as the left and right syntactic components (via *left* and *right*, respectively), as follows:

$$\begin{aligned}
\mathit{ValueSubstitution} \triangleq {} & (\mathit{TypeVariable} \rightharpoonup \mathit{Type}) \\
& \times (\mathit{Variable} \rightharpoonup \mathit{Value}) \\
& \times (\mathit{CoVariable} \rightharpoonup \mathit{CoValue}) \\
\mathit{sem} : {} & \mathit{Substitution} \to \mathit{TypeSubstitution} \\
\mathit{left}, \mathit{right} : {} & \mathit{Substitution} \to \mathit{ValueSubstitution}
\end{aligned}$$

We begin with the most fundamental interpretation of sequents, which is on the judgement $\left(\Gamma \vdash_{\mathcal{G}}^{\Theta} \Delta\right) \mathbf{seq}$ that a sequent is well-formed. This well-formedness judgement is interpreted as the set of possible substitutions that map each variable mentioned in $\left(\Gamma \vdash_{\mathcal{G}}^{\Theta} \Delta\right) \mathbf{seq}$ to an appropriate semantic entities dictated by the previous interpretations:

$$\begin{aligned}
& \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right) \mathbf{seq} \\
& \triangleq \{\sigma \in \mathit{Substitution} \\
& \quad \mid \forall \mathit{decl} \in \mathcal{G}, \mathit{decl} \text{ } \mathit{defines} \text{ } \mathsf{F}(\overrightarrow{X : k}) : \mathcal{S} \implies \pi_2(\sigma(\mathsf{F})) = [\![\mathit{decl}]\!]_{\mathit{sem}(\sigma)} \in \left[\!\!\left[\overrightarrow{k \to \mathcal{S}}\right]\!\!\right]_{\mathit{sem}(\sigma)} \\
& \quad \wedge \forall X{:}k \in \Theta, [\![k]\!]_{\mathit{sem}(\sigma)} \in \mathit{SemKind} \wedge \sigma(X) \in \pi_2([\![k]\!]_{\mathit{sem}(\sigma)}) \\
& \quad \wedge \forall x{:}A \in \Gamma, \exists \mathcal{S}, \sigma(x) \in [\![A]\!]_{\mathit{sem}(\sigma)} \in \mathit{SemType}(\mathbb{T}_{\mathcal{S}}) \\
& \quad \wedge \forall \alpha{:}A \in \Delta, \exists \mathcal{S}, \sigma(\alpha) \in [\![A]\!]_{\mathit{sem}(\sigma)} \in \mathit{SemType}(\mathbb{T}_{\mathcal{S}})\}
\end{aligned}$$

Intuitively, finding a particular $\sigma \in \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right) \mathbf{seq}$ is the semantic evidence that the sequent is coherent enough to reason about. That's why the interpretation of every other sequent that follows is predicated upon finding such evidence before any firm statements are made.

The sequents for the sorting judgement affirms that the interpretations of the kind is indeed the stated sort, assuming that there is a well-formed substitution for the sequent. The sequents for the kinding judgement affirms that, under any well-formed substitution, the syntactic type is related to the semantic interpretation by its kind. Furthermore, the sequents for type conversion require that both types be semantically well-kinded by the previous interpretation, and also that the semantic interpretation of the types are the same. These are formally defined as follows:

$$\begin{aligned}
\mathcal{G} \vDash \mathit{decl} &\triangleq \forall \sigma \in \left(\vDash_{\mathcal{G}}\right) \mathbf{seq}\, , [\![\mathit{decl}]\!]_{\mathit{sem}(\sigma)} \in \mathit{Universe} \\
\Theta \vDash_{\mathcal{G}} k : s &\triangleq \forall \sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}\, , [\![k]\!]_{\mathit{sem}(\sigma)} \in [\![s]\!] \\
\Theta \vDash_{\mathcal{G}} A : k &\triangleq \forall \sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}\, , [\![k]\!]_{\mathit{sem}(\sigma)} \in \mathit{SemKind} \implies \\
&\qquad A\{\mathit{left}(\sigma)\}\ [\![k]\!]_{\mathit{sem}(\sigma)}\ [\![A]\!]_{\mathit{sem}(\sigma)} \\
\Theta \vDash_{\mathcal{G}} A = B : k &\triangleq \forall \sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}\, , [\![k]\!]_{\mathit{sem}(\sigma)} \in \mathit{SemKind} \implies \\
&\qquad A\{\mathit{left}(\sigma)\}\ [\![k]\!]_{\mathit{sem}(\sigma)}\ [\![A]\!]_{\mathit{sem}(\sigma)} \wedge B\{\mathit{right}(\sigma)\}\ [\![k]\!]_{\mathit{sem}(\sigma)}\ [\![B]\!]_{\mathit{sem}(\sigma)} \\
&\qquad \wedge [\![A]\!]_{\mathit{sem}(\sigma)} = [\![B]\!]_{\mathit{sem}(\sigma)}
\end{aligned}$$

where $\{\mathit{left}(\sigma)\}$ and $\{\mathit{right}(\sigma)\}$ is notation for applying $\mathit{right}(\sigma)$ as the simultaneous capture-avoiding substitution of all type variables in its domain.

We now get down to the typing judgements for programs (i.e. commands and (co-)terms). First, we define what it means for two commands, terms, or co-terms to be semantically related to one another with respect to some sequent as follows:

$$\begin{aligned}
c \Leftrightarrow c' : \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right) &\triangleq \forall \sigma \in \left(\Gamma \vDash_{\mathcal{G}} \Delta\right) \mathbf{seq}\, , c\{\mathit{left}(\sigma)\} \perp\!\!\!\perp c'\{\mathit{right}(\sigma)\} \\
\Gamma \vDash^{\Theta}_{\mathcal{G}} v \Leftrightarrow v' : A \mid \Delta &\triangleq \forall \sigma \in \left(\Gamma \vDash_{\mathcal{G}} \Delta\right) \mathbf{seq}\, , \forall \mathcal{S}, A\{\mathit{left}(\sigma)\}\ [\![\mathcal{S}]\!]_{\pi_2 \circ \sigma}\ [\![A]\!]_{\mathit{sem}(\sigma)} \implies \\
&\qquad v\{\mathit{left}(\sigma)\}\ [\![A]\!]_{\mathit{sem}(\sigma)}\ v'\{\mathit{right}(\sigma)\} \\
\Gamma \mid e \Leftrightarrow e' : A \vDash^{\Theta}_{\mathcal{G}} \Delta &\triangleq \forall \sigma \in \left(\Gamma \vDash_{\mathcal{G}} \Delta\right) \mathbf{seq}\, , \forall \mathcal{S}, A\{\mathit{left}(\sigma)\}\ [\![\mathcal{S}]\!]_{\pi_2 \circ \sigma}\ [\![A]\!]_{\mathit{sem}(\sigma)} \implies \\
&\qquad e\{\mathit{left}(\sigma)\}\ [\![A]\!]_{\mathit{sem}(\sigma)}\ e'\{\mathit{right}(\sigma)\}
\end{aligned}$$

Intuitively, the semantic judgement that two commands are related means that, under the left and right components of any well-formed substitution, they are related by $\perp\!\!\!\perp$. Likewise the semantic judgement that two terms or co-terms are related means that (under the left and right components of any well-formed substitution), they are related by the interpretation of the type they inhabit. Furthermore, we give the semantic judgement that commands and (co-)terms are well-typed in terms of the above by affirming they are related to themselves.

$$c : \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right) \triangleq c \Leftrightarrow c : \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)$$
$$\Gamma \vDash_{\mathcal{G}}^{\Theta} v : A \mid \Delta \triangleq \Gamma \vDash_{\mathcal{G}}^{\Theta} v \Leftrightarrow v : A \mid \Delta$$
$$\Gamma \mid e : A \vDash_{\mathcal{G}}^{\Theta} \Delta \triangleq \Gamma \mid e \Leftrightarrow e : A \vDash_{\mathcal{G}}^{\Theta} \Delta$$

Adequacy

We have built a model for the parametric $\mu\tilde{\mu}$-calculus, but we have not yet seen that it corresponds at all to the statics of the language. In other words, we need to demonstrate *adequacy*: derivations of a syntactic judgement implies the truth of the corresponding semantic judgement. With this in mind, we now show the adequacy of the model via the following fundamental lemmas.

The first fundamental lemma demonstrates the adequacy of the sorting and kinding rules.

Lemma 7.7 (Kind adequacy)**.** *For any focalizing strategies $\vec{\mathcal{S}}$, safety condition $\mathbb{S}$ and family of $\mathbb{S}$-worlds $\mathbb{T}_{\mathcal{S}}$, and size measurement,*

a) if $\mathcal{G} \vdash decl$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\mathcal{G} \vDash decl$,

b) if $\Theta \vdash_{\mathcal{G}} k : s$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Theta \vDash_{\mathcal{G}} k : s$,

c) if $\Theta \vdash_{\mathcal{G}} A : k$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Theta \vDash_{\mathcal{G}} A : k$, and

d) if $\Theta \vdash_{\mathcal{G}} A = B : k$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Theta \vDash_{\mathcal{G}} A = B : k$.

Proof. By mutual induction on the derivations, demonstrating that each inference rule I is *sound*: the interpretation of its conclusion follows from the interpretation of the premises, giving the semantic version of the rule $[\![I]\!]$.

The interpretation of the sorting rules are sound as follows:

- To show $\Theta \vDash_{\mathcal{G}} \mathcal{S} : \Box$, note that $\llbracket \mathcal{S} \rrbracket_\sigma = (\mathit{SemType}(\mathbb{T}_{\mathcal{S}}), \mathit{Type} \times \mathit{SemType}(\mathbb{T}_{\mathcal{S}})) \in \mathit{SemKind}$ for any σ and that for all $A \twoheadrightarrow_\beta A' \in \mathit{Type}$ and $\mathbb{A} \in \mathit{SemType}(\mathbb{T}_{\mathcal{S}})$, both $A \llbracket \mathcal{S} \rrbracket_\sigma \mathbb{A}$ and $A' \llbracket \mathcal{S} \rrbracket_\sigma \mathbb{A}$ since all syntactic and semantic types are related by $\llbracket \mathcal{S} \rrbracket_\sigma$, so $\llbracket \mathcal{S} \rrbracket_\sigma \in \llbracket \Box \rrbracket$.

- To show $\Theta \vDash_{\mathcal{G}} \mathsf{Ix} : \blacksquare$, note that $\llbracket \mathsf{Ix} \rrbracket_\sigma = \Big(\mathbb{N}, \{(M, \mathbb{M}) \mid M \sim^{\mathbb{N}} \mathbb{M}\}\Big) \in \mathit{SemKind}$ for any σ.

- To show $\Theta \vDash_{\mathcal{G}} \mathsf{Ord} : \Box$, note that

  $$\llbracket \mathsf{Ord} \rrbracket_\sigma = \Big(\mathbb{O}, \{(M, \mathbb{M}) \mid \exists M' \in \mathit{Type}.M \twoheadrightarrow_\beta M' \sim^{\mathbb{O}} \mathbb{M}' \leq \mathbb{M}\}\Big) \in \mathit{SemKind}$$

  for any σ, and that the syntactic-semantic relation of $\llbracket \mathsf{Ord} \rrbracket_\sigma$ is closed under β expansion by definition.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} k : \blacksquare \quad \Theta \vDash_{\mathcal{G}} l : \Box}{\Theta \vDash_{\mathcal{G}} k \to l : \Box}$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}$, so that we know $\llbracket k \rrbracket_\sigma \in \blacksquare$ and $\llbracket l \rrbracket_{\mathit{sem}(\sigma)} \in \Box$ from the premises. Thus, $\llbracket k \to l \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$, and the syntactic-semantic relation of $\llbracket k \to l \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$ is closed under β expansion because the syntact-semantic relation of $\llbracket l \rrbracket_{\mathit{sem}(\sigma)}$ is: if $A \twoheadrightarrow_\beta A'$ and $A' \llbracket k \to l \rrbracket_{\mathit{sem}(\sigma)} \mathbb{A}$ then for all $B \llbracket k \rrbracket_{\mathit{sem}(\sigma)} \mathbb{B}$ it follows that $A'\ B \llbracket l \rrbracket_{\mathit{sem}(\sigma)} \mathbb{A}(\mathbb{B})$, so that $A\ B \llbracket l \rrbracket_{\mathit{sem}(\sigma)} \mathbb{A}(\mathbb{B})$ as well because $\llbracket l \rrbracket_{\mathit{sem}(\sigma)} \in \llbracket \Box \rrbracket$.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} M : \mathsf{Ord}}{\Theta \vDash_{\mathcal{G}} < M : \Box}$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}$, so that we know $\llbracket M \rrbracket_{\mathit{sem}(\sigma)} \in \llbracket \mathsf{Ord} \rrbracket_{\mathit{sem}(\sigma)}$ from the premise. Thus, the domain $\{\mathbb{M}' \in \mathbb{O} \mid \mathbb{M}' < \llbracket M \rrbracket_{\mathit{sem}(\sigma)}\}$ of $\llbracket < M \rrbracket_{\mathit{sem}(\sigma)}$ is a defined subset of $\mathbb{O}$, and the syntactic-semantic relation of $\llbracket M \rrbracket_{\mathit{sem}(\sigma)}$ is closed under β expansion for the same reason that $\llbracket \mathsf{Ord} \rrbracket_{\mathit{sem}(\sigma)}$ is.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} k : \Box}{\Theta \vDash_{\mathcal{G}} k : \blacksquare}$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}$, so that we know $\llbracket k \rrbracket_{\mathit{sem}(\sigma)} \in \llbracket \Box \rrbracket$ from the premise, and note that this implies that $\llbracket k \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind} = \llbracket \blacksquare \rrbracket$.

The interpretation of the kinding rules are sound as follows:

- To show $\dfrac{(X : k) \notin \Theta'}{\Theta, X : k, \Theta' \vDash_{\mathcal{G}} X : k} \llbracket TV \rrbracket$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}$ and $\daleth[\mathit{sem}(\sigma)] \in \mathit{SemKind}$, and note that

  $$(X\left\{\mathit{left}(\sigma)\right\}, \llbracket X \rrbracket_{(\sigma)}) = (\mathit{left}(\sigma)(X), \mathit{sem}(\sigma)(X)) = \sigma(X) \in \pi_2(\llbracket k \rrbracket_\sigma)$$

- To show $\dfrac{\overrightarrow{\Theta \vDash_{\mathcal{G}} C : k} \quad (\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}) \in \mathcal{G}}{\Theta \vDash_{\mathcal{G}} \mathsf{F}(\vec{C}) : \mathcal{S}}\ \llbracket \mathsf{F}T \rrbracket$ suppose that $\sigma \in \left(\vDash_{\mathcal{G}}^{\Theta}\right)\mathbf{seq}$, so we know that $\overrightarrow{C\,\{\mathit{left}(\sigma)\}\ \llbracket k\rrbracket_{\mathit{sem}(\sigma)}\ \llbracket C\rrbracket_{\mathit{sem}(\sigma)}}$ from the premise. Since $(\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}) \in \mathcal{G}$, we also know that for the $\mathit{decl} \in \mathcal{G}$ defining $\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}$, we have

  $$\mathit{sem}(\sigma)(\mathsf{F}) = \llbracket \mathit{decl}\rrbracket_{\mathit{sem}(\sigma)} \in \left\llbracket \overrightarrow{k \to} \mathcal{S} \right\rrbracket_{\mathit{sem}(\sigma)} = \overrightarrow{\pi_1(\llbracket k\rrbracket_{\mathit{sem}(\sigma)}) \to} \mathit{SemType}(\mathbb{T}_{\mathcal{S}})$$

  Thus, $\mathsf{F}(\vec{C})\,\{\mathit{left}(\sigma)\} = \mathsf{F}(\overrightarrow{C\,\{\sigma\}})\ \llbracket S\rrbracket_{\mathit{sem}(\sigma)}\ \left\llbracket \mathsf{F}(\vec{C}) \right\rrbracket_{\mathit{sem}(\sigma)} = \mathit{sem}(\sigma)(\mathsf{F})(\overrightarrow{\llbracket C\rrbracket_{\mathit{sem}(\sigma)}})$.

- To show $\dfrac{\Theta, X : k \vDash_{\mathcal{G}} A : l}{\Theta \vDash_{\mathcal{G}} \lambda X : k.A : k \to l}\ \llbracket {\to} I^2 \rrbracket$ suppose that $\sigma \in \left(\vDash_{\mathcal{G}}^{\Theta}\right)\mathbf{seq}$ and $\llbracket k \to l\rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$, so we know that $\llbracket k\rrbracket_{\mathit{sem}(\sigma)}, \llbracket l\rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$ and thus $A\,\{\mathit{left}(\sigma), B/X\}\ \llbracket l\rrbracket_{\mathit{sem}(\sigma)\mathbb{B}/X}\ \llbracket A\rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X}$ for all $B\ \llbracket k\rrbracket_{\mathit{sem}(\sigma)}\ \mathbb{B}$ from the premise. Note that

  $$(\lambda X{:}k.A)\,\{\mathit{left}(\sigma)\}\ \ B \to_{\beta} A\,\{\mathit{left}(\sigma), B/X\} \quad \llbracket \lambda X{:}k.A\rrbracket_{\mathit{sem}(\sigma)}(\mathbb{B}) = \llbracket A\rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X}$$

  Thus, $(\lambda X{:}k.A)\,\{\mathit{left}(\sigma)\}\ \llbracket k \to l\rrbracket_{\mathit{sem}(\sigma)}\ \llbracket \lambda X{:}k.A\rrbracket_{\mathit{sem}(\sigma)}$.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} A : k \to l \quad \Theta \vDash_{\mathcal{G}} B : k \quad \Theta \vDash_{\mathcal{G}} k \to l : \Box}{\Theta \vDash_{\mathcal{G}} A\ B : l}\ \llbracket {\to} E^2 \rrbracket$ suppose that $\sigma \in \left(\vDash_{\mathcal{G}}^{\Theta}\right)\mathbf{seq}$, so we know that $\llbracket k \to l\rrbracket_{\mathit{sem}(\sigma)} \in \llbracket \Box \rrbracket \subseteq \mathit{SemKind}$ (which implies $\llbracket k\rrbracket_{\mathit{sem}(\sigma)} \in \llbracket \blacksquare \rrbracket = \mathit{SemKind}$), $A\,\{\mathit{left}(\sigma)\}\ \llbracket k \to l\rrbracket_{\mathit{sem}(\sigma)}\ \llbracket A\rrbracket_{\mathit{sem}(\sigma)}$, and $B\,\{\mathit{left}(\sigma)\}\ \llbracket k\rrbracket_{\mathit{sem}(\sigma)}\ \llbracket B\rrbracket_{\mathit{sem}(\sigma)}$ from the premises. It follows that $(A\ B)\,\{\mathit{left}(\sigma)\}\ \llbracket l\rrbracket_{\mathit{sem}(\sigma)}\ \llbracket A\rrbracket_{\mathit{sem}(\sigma)}(\llbracket B\rrbracket_{\mathit{sem}(\sigma)}) = \llbracket A\ B\rrbracket_{\mathit{sem}(\sigma)}$ by the definition of $\llbracket k \to l\rrbracket_{\mathit{sem}(\sigma)}$.

- To show $\Theta \vDash_{\mathcal{G}} 0 : \mathsf{Ix}$ note that $0\,\{\sigma_1\} = 0$, $\llbracket 0\rrbracket_{\sigma_2} = 0 \in \mathbb{N}$, and $0\ \llbracket \mathsf{Ix}\rrbracket_{\sigma_2}\ 0$ for any substitutions σ_1 and σ_2.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} M : \mathsf{Ix}}{\Theta \vDash_{\mathcal{G}} M + 1 : \mathsf{Ix}}$ suppose that $\sigma \in \left(\vDash_{\mathcal{G}}^{\Theta}\right)\mathbf{seq}$, so we know that $M\,\{\mathit{left}(\sigma)\}\ \llbracket \mathsf{Ix}\rrbracket_{\mathit{sem}(\sigma)}\ \llbracket M\rrbracket_{\mathit{sem}(\sigma)}$ from the premise. It follows by definition of $\llbracket \mathsf{Ix}\rrbracket_{\mathit{sem}(\sigma)}$ that $\llbracket M\rrbracket_{\mathit{sem}(\sigma)} \in \mathbb{N}$ and $M\,\{\mathit{left}(\sigma)\} \sim^{\mathbb{N}} \llbracket M\rrbracket_{\mathit{sem}(\sigma)}$, so

  $$+1(M)\,\{\mathit{left}(\sigma)\}\ \llbracket \mathsf{Ix}\rrbracket_{\mathit{sem}(\sigma)}\ \llbracket +1(M)\rrbracket_{\mathit{sem}(\sigma)}$$

- To show $\Theta \vDash_{\mathcal{G}} \infty : \mathsf{Ord}$ note that $\infty\,\{\sigma_1\} = \infty$, $\llbracket \infty \rrbracket_{\sigma_2} = \infty \in \mathbb{O}$, and $\infty\ \llbracket \mathsf{Ord} \rrbracket_{\sigma_2}\ \infty$ for any substitutions σ_1 and σ_2.

- To show $\Theta \vDash_{\mathcal{G}} 0 : < \infty$ note that $0\,\{\sigma_1\} = 0$, $\llbracket 0 \rrbracket_{\sigma_2} = 0 \in \mathbb{O}$, and $0 < \infty$ is guaranteed by the size measurement for any substitutions σ_1 and σ_2.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} M : < \infty}{\Theta \vDash_{\mathcal{G}} M + 1 : < \infty}$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right)\mathbf{seq}$, so we know that $M\,\{\mathit{left}(\sigma)\}\ \llbracket < \infty \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket M \rrbracket_{\mathit{sem}(\sigma)}$ from the premise. The fact that $+1(\llbracket M \rrbracket_{\mathit{sem}(\sigma)}) < \infty$, and thus that $+1(M)\,\{\mathit{left}(\sigma)\}\ \llbracket < \infty \rrbracket_{\mathit{sem}(\sigma)}\ +1(\llbracket M \rrbracket_{\mathit{sem}(\sigma)})$, is guaranteed by the size measurement which forces ∞ to be a limit of $+1$.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} N : \mathsf{Ord} \quad \Theta \vDash_{\mathcal{G}} M : < N}{\Theta \vDash_{\mathcal{G}} M : < M + 1}$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right)\mathbf{seq}$ and $\llbracket < M + 1 \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$, so we know that $N\,\{\mathit{left}(\sigma)\}\ \llbracket \mathsf{Ord} \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket N \rrbracket_{\mathit{sem}(\sigma)}$ (which implies that $\llbracket < N \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{skind}$) and $M\,\{\mathit{left}(\sigma)\}\ \llbracket < N \rrbracket_{\sigma}\ \llbracket M \rrbracket_{\mathit{sem}(\sigma)}$ from the premises. The size measurement guarantees forces $+1$ to be strictly increasing, so $\llbracket M \rrbracket_{\mathit{sem}(\sigma)} < +1(\llbracket M \rrbracket_{\mathit{sem}(\sigma)}) = \llbracket M + 1 \rrbracket_{\mathit{sem}(\sigma)}$. Thus $M\,\{\mathit{left}(\sigma)\}\ \llbracket < M \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket M \rrbracket_{\mathit{sem}(\sigma)}$.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} M : < M' \quad \Theta \vDash_{\mathcal{G}} M' : < N}{\Theta \vDash_{\mathcal{G}} M : < N}$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right)\mathbf{seq}$ and $\llbracket < N \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$, so we know that $M'\,\{\mathit{left}(\sigma)\}\ \llbracket < N \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket M' \rrbracket_{\mathit{sem}(\sigma)}$ (which implies $\llbracket < M' \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$) and $M\,\{\mathit{left}(\sigma)\}\ \llbracket < M' \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket M \rrbracket_{\mathit{sem}(\sigma)}$. Note that $\llbracket M \rrbracket_{\mathit{sem}(\sigma)} < \llbracket M' \rrbracket_{\mathit{sem}(\sigma)} < \llbracket N \rrbracket_{\mathit{sem}(\sigma)}$ by definition, so $\llbracket M \rrbracket_{\mathit{sem}(\sigma)} < \llbracket N \rrbracket_{\mathit{sem}(\sigma)}$ is forced by transitivity of the size measurement, and thus $M\,\{\mathit{left}(\sigma)\}\ \llbracket < N \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket M \rrbracket_{\mathit{sem}(\sigma)}$.

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} N : \mathsf{Ord} \quad \Theta \vDash_{\mathcal{G}} M : < N}{\Theta \vDash_{\mathcal{G}} M : \mathsf{Ord}}$

suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right)\mathbf{seq}$, so we know that $N\,\{\mathit{left}(\sigma)\}\ \llbracket \mathsf{Ord} \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket N \rrbracket_{\mathit{sem}(\sigma)}$ (which implies that $\llbracket < N \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{skind}$) and $M\,\{\mathit{left}(\sigma)\}\ \llbracket < N \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket M \rrbracket_{\mathit{sem}(\sigma)}$ from the premises. $M\,\{\mathit{left}(\sigma)\}\ \llbracket \mathsf{Ord} \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket M \rrbracket_{\mathit{sem}(\sigma)}$ follows from the inclusion of the set $\mathbb{M} \in \mathbb{O} \mid \mathbb{M} < \llbracket N \rrbracket_{\mathit{sem}(\sigma)}$ in $\mathbb{O}$.

The interpretation of $\beta\eta$ conversion of types are sound as follows:

- To show $\dfrac{\Theta, X : k \vDash_{\mathcal{G}} A : l \quad \Theta \vDash_{\mathcal{G}} B : k \quad \Theta \vDash_{\mathcal{G}} k \to l : \Box}{\Theta \vDash_{\mathcal{G}} (\lambda X{:}k.A)\ B =_{\beta\eta} A\{B/X\} : l}\ [\![\beta]\!]$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}$ so we know that for all C $[\![k]\!]_{sem(\sigma)}$ $\mathbb{C}$

$$\begin{aligned} [\![k \to l]\!]_{sem(\sigma)} &\in [\![\Box]\!] \subseteq \mathit{SemKind} \\ [\![k]\!]_{sem(\sigma)} &\in [\![\blacksquare]\!] = \mathit{SemKind} \\ [\![l]\!]_{sem(\sigma)} &\in [\![\Box]\!] \subseteq \mathit{SemKind} \\ B\{\mathit{left}(\sigma)\}\ [\![k]\!]_{sem(\sigma)}&\ [\![B]\!]_{sem(\sigma)} \\ A\{\mathit{left}(\sigma), C/X\}\ [\![k \to l]\!]_{sem(\sigma)}&\ [\![A]\!]_{sem(\sigma),\mathbb{C}/X} \end{aligned}$$

from the premises. Thus, it follows that

$$\begin{aligned} ((\lambda X{:}k.A)\ B)\{\mathit{left}(\sigma)\} &\to_\beta A\{\mathit{left}(\sigma), B/X\} \\ [\![(\lambda X{:}k.A)\ B]\!]_{sem(\sigma)} &= [\![A]\!]_{sem(\sigma),[\![B]\!]_{sem(\sigma)}/X} \in [\![\Box]\!] \\ A\{\mathit{left}(\sigma), B\{\mathit{left}(\sigma)\}/X\}\ [\![l]\!]_{sem(\sigma)}&\ [\![A]\!]_{sem(\sigma),[\![B]\!]_{sem(\sigma)}/X} \\ ((\lambda X{:}k.A)\ B)\{\mathit{left}(\sigma)\}\ [\![l]\!]_{sem(\sigma)}&\ [\![A]\!]_{sem(\sigma),[\![B]\!]_{sem(\sigma)}/X} = [\![(\lambda X{:}k.A)\ B]\!]_{sem(\sigma)} \end{aligned}$$

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} A : k \to l}{\Theta \vDash_{\mathcal{G}} \lambda X{:}k.A\ X =_{\beta\eta} A : k \to l}\ [\![\eta]\!]$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}$ and $[\![ktol]\!]_{sem(\sigma)} \in \mathit{SemKind}$ so we know that $A\{\mathit{left}(\sigma)\}$ $[\![k \to l]\!]_{sem(\sigma)}$ $[\![A]\!]_{sem(\sigma)}$ from the premise. It follows that $(\lambda X{:}k.A\ X)\{\mathit{left}(\sigma)\}$ $[\![k \to l]\!]_{sem(\sigma)}$ $[\![\lambda X{:}k.A\ X]\!]_{sem(\sigma)} = [\![A]\!]_{sem(\sigma)}$ by the definition of $[\![k \to l]\!]_{sem(\sigma)}$.

- The closure rule $\dfrac{\Theta \vDash_{\mathcal{G}} A : k}{\Theta \vDash_{\mathcal{G}} A =_{\beta\eta} A : k}\ [\![\mathit{refl}]\!]$ follows immediately from the premise.

- The closure rule $\dfrac{\Theta \vDash_{\mathcal{G}} B =_{\beta\eta} A : k}{\Theta \vDash_{\mathcal{G}} A =_{\beta\eta} B : k}\ [\![\mathit{symm}]\!]$ follows immediately from the premise.

- The closure rule $\dfrac{\Theta \vDash_{\mathcal{G}} A =_{\beta\eta} B : k \quad \Theta \vDash_{\mathcal{G}} B =_{\beta\eta} C : k}{\Theta \vDash_{\mathcal{G}} A =_{\beta\eta} C : k}\ [\![\mathit{trans}]\!]$ follows immediately from the premise.

– To show

$$\frac{\Theta \vDash_{\mathcal{G}} \mathsf{F}(\vec{C}) : \mathcal{S} \quad \overrightarrow{\Theta \vDash_{\mathcal{G}} C =_{\beta\eta} C' : k} \quad \Theta \vDash_{\mathcal{G}} \mathsf{F}(\vec{C}) : \mathcal{S} \quad (\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}) \in \mathcal{G}}{\Theta \vDash_{\mathcal{G}} \mathsf{F}(\vec{C}) =_{\beta\eta} \mathsf{F}(\vec{C}) : \mathcal{S}} \; \llbracket \mathsf{F}T \rrbracket$$

suppose that $\sigma \in \left(\vDash_{\mathcal{G}}^{\Theta}\right) \mathbf{seq}$. Note that $\mathsf{F}(\vec{C})\,\{\mathit{left}(\sigma)\}\; \llbracket \mathcal{S} \rrbracket_{\mathit{sem}(\sigma)}\; \left\llbracket \mathsf{F}(\vec{C}) \right\rrbracket_{\mathit{sem}(\sigma)}$ and $\mathsf{F}(\vec{C'})\,\{\mathit{left}(\sigma)\}\; \llbracket \mathcal{S} \rrbracket_{\mathit{sem}(\sigma)}\; \left\llbracket \mathsf{F}(\vec{C'}) \right\rrbracket_{\mathit{sem}(\sigma)}$ follow immediately from the first and last premises. We also learn from $(\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}) \in \mathcal{G}$ that $\overrightarrow{\llbracket k \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}}$, so from the remaining premises, we have that $\overrightarrow{\llbracket C \rrbracket_{\mathit{sem}(\sigma)} = \llbracket C' \rrbracket_{\mathit{sem}(\sigma)}}$, and thus

$$\left\llbracket \mathsf{F}(\vec{C}) \right\rrbracket_{\mathit{sem}(\sigma)} = \sigma(\mathsf{F})(\overrightarrow{\llbracket C \rrbracket_{\mathit{sem}(\sigma)}}) = \sigma(\mathsf{F})(\overrightarrow{\llbracket C' \rrbracket_{\mathit{sem}(\sigma)}}) = \left\llbracket \mathsf{F}(\vec{C'}) \right\rrbracket_{\mathit{sem}(\sigma)}$$

– To show $\dfrac{\Theta, X : k \vDash_{\mathcal{G}} A =_{\beta\eta} A' : l}{\Theta \vDash_{\mathcal{G}} \lambda X{:}k.A =_{\beta\eta} \lambda X{:}k.A' : k \to l} \; \llbracket \to I^2 \rrbracket$ suppose that $\sigma \in \left(\vDash_{\mathcal{G}}^{\Theta}\right) \mathbf{seq}$ and $\llbracket k \to l \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$ (which implies that $\llbracket k \rrbracket_{\mathit{sem}(\sigma)}, \llbracket l \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemKind}$), so from the premise, we know that for all $B\; \llbracket k \rrbracket_{\mathit{sem}(\sigma)}\; \mathbb{B}$

$$\begin{gathered}
A\,\{\mathit{left}(\sigma), B/X\}\; \llbracket l \rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X}\; \llbracket A \rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X} \\
A'\,\{\mathit{left}(\sigma), B/X\}\; \llbracket l \rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X}\; \llbracket A' \rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X} \\
\llbracket A \rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X} = \llbracket A' \rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X}
\end{gathered}$$

Note also that for all $B\; \llbracket k \rrbracket_{\mathit{sem}(\sigma)}\; \mathbb{B}$

$$\begin{gathered}
(\lambda X{:}k.A)\,\{\mathit{left}(\sigma)\}\;\; B \to_{\beta} A\,\{\mathit{left}(\sigma), B/X\} \\
(\lambda X{:}k.A')\,\{\mathit{left}(\sigma)\}\;\; B \to_{\beta} A'\,\{\mathit{left}(\sigma), B/X\} \\
\llbracket \lambda X{:}k.A \rrbracket_{\mathit{sem}(\sigma)}(\mathbb{B}) = \llbracket A \rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X} \\
\llbracket \lambda X{:}k.A' \rrbracket_{\mathit{sem}(\sigma)}(\mathbb{B}) = \llbracket A' \rrbracket_{\mathit{sem}(\sigma),\mathbb{B}/X}
\end{gathered}$$

Thus, it follows that

$$\begin{gathered}
\llbracket \lambda X{:}k.A \rrbracket_{\mathit{sem}(\sigma)} = \llbracket \lambda X{:}k.A' \rrbracket_{\mathit{sem}(\sigma)} \\
(\lambda X{:}k.A)\,\{\mathit{left}(\sigma)\} \to_{\beta} \llbracket \lambda X{:}k.A \rrbracket_{\mathit{sem}(\sigma)} \\
(\lambda X{:}k.A')\,\{\mathit{left}(\sigma)\} \to_{\beta} \llbracket \lambda X{:}k.A' \rrbracket_{\mathit{sem}(\sigma)}
\end{gathered}$$

- To show $\dfrac{\Theta \vDash_{\mathcal{G}} A =_{\beta\eta} A' : k \to l \quad \Theta \vDash_{\mathcal{G}} B =_{\beta\eta} B' : k \quad \Theta \vDash_{\mathcal{G}} k \to l : \Box}{\Theta \vDash_{\mathcal{G}} A\ B =_{\beta\eta} A'\ B' : l} \llbracket {\to}E^2 \rrbracket$ suppose that $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right) \mathbf{seq}$, so we know that

$$\begin{array}{cc}
\llbracket k \to l \rrbracket_{sem(\sigma)} \in \llbracket \Box \rrbracket \subseteq \mathit{SemKind} & \llbracket k \rrbracket_{sem(\sigma)} \in \mathit{SemKind} \\
A \{\mathit{left}(\sigma)\} \llbracket k \to l \rrbracket_{sem(\sigma)} \llbracket A \rrbracket_{sem(\sigma)} & B \{\mathit{left}(\sigma)\} \llbracket k \rrbracket_{sem(\sigma)} \llbracket B \rrbracket_{sem(\sigma)} \\
A' \{\mathit{left}(\sigma)\} \llbracket k \to l \rrbracket_{sem(\sigma)} \llbracket A' \rrbracket_{sem(\sigma)} & B' \{\mathit{left}(\sigma)\} \llbracket k \rrbracket_{sem(\sigma)} \llbracket B' \rrbracket_{sem(\sigma)} \\
\llbracket A \rrbracket_{sem(\sigma)} = \llbracket A' \rrbracket_{sem(\sigma)} & \llbracket B \rrbracket_{sem(\sigma)} = \llbracket B' \rrbracket_{sem(\sigma)}
\end{array}$$

  from the premises. It follows from the definition of $\llbracket k \to l \rrbracket_{sem(\sigma)}$ and the fact that $(A\ B) \{\mathit{left}(\sigma)\} = A \{\mathit{left}(\sigma)\}\ B \{\mathit{left}(\sigma)\}$ and $(A'\ B') \{\mathit{left}(\sigma)\} = A' \{\mathit{left}(\sigma)\}\ B' \{\mathit{left}(\sigma)\}$ that

$$\begin{gathered}
(A\ B) \{\mathit{left}(\sigma)\} \llbracket l \rrbracket_{sem(\sigma)} \llbracket A\ B \rrbracket_{\mathit{left}(\sigma)} \\
(A'\ B') \{\mathit{left}(\sigma)\} \llbracket l \rrbracket_{sem(\sigma)} \llbracket A'\ B' \rrbracket_{\mathit{left}(\sigma)} \\
\llbracket A \rrbracket_{sem(\sigma)}(\llbracket B \rrbracket_{sem(\sigma)}) = \llbracket A' \rrbracket_{sem(\sigma)}(\llbracket B' \rrbracket_{sem(\sigma)})
\end{gathered}$$

- The other inference rules for converting inside the context of a successor Ord type index follow similarly.

The soundness of the interpretation of the well-formedness of declaration rules follows immediately from the definition of the interpretation of declarations and the premise. □

Turning now to the typing rules for programs and program equality, we have that there is an unfortunate redundancy of rules. In particular, because the *compatibility* of the type equality relation quantifies over all contexts, it effectively duplicates all the typing rules. Additionally, quantifying over arbitrary contexts adds extra complication to any direct proof. Therefore, instead of handling typed equality as is, we resort to an equivalent rephrasing of the relation that effectively combines both the typing and compatibility rules into a single form as shown in Figures 7.1 and 7.2. As stated, this alternative definition effectively subsumes *both* the derivations of syntactic equality and typing, so we can focus on just $c \Leftrightarrow_R c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ (et al.) instead of both the $c =_R c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ and $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$.

Lemma 7.8. *a)* $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ *is derivable if and only if* $c \Leftrightarrow_R c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ *is.*

Conversion rules:

$$\frac{c \Leftrightarrow_R c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right) \quad c \succ_R c' \quad c' \Leftrightarrow_R c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)}{c \Leftrightarrow_R c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)} \succ$$

$$\frac{c \Leftrightarrow_R c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right) \quad c \prec_R c' \quad c' \Leftrightarrow_R c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)}{c' \Leftrightarrow_R c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)} \prec$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v : A \mid \Delta \quad v \succ_R v' \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v' \Leftrightarrow_R v' : A \mid \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v' : A \mid \Delta} \succ R$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v : A \mid \Delta \quad v \prec_R v' \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v' \Leftrightarrow_R v' : A \mid \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v' \Leftrightarrow_R v : A \mid \Delta} \prec R$$

$$\frac{\Gamma \mid e \Leftrightarrow_R e : A \vdash^{\Theta}_{\mathcal{G}} \Delta \quad e \succ_R e' \quad \Gamma \mid e' \Leftrightarrow_R e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta}{\Gamma \mid e \Leftrightarrow_R e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta} \succ L$$

$$\frac{\Gamma \mid e \Leftrightarrow_R e : A \vdash^{\Theta}_{\mathcal{G}} \Delta \quad e \prec_R e' \quad \Gamma \mid e' \Leftrightarrow_R e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta}{\Gamma \mid e' \Leftrightarrow_R e : A \vdash^{\Theta}_{\mathcal{G}} \Delta} \succ L$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v' : A \mid \Delta \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v' \Leftrightarrow_R v'' : A \mid \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v'' : A \mid \Delta} \; \mathit{transR}$$

$$\frac{\Gamma \mid e \Leftrightarrow_R e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta \quad \Gamma \mid e' \Leftrightarrow_R e'' : A \vdash^{\Theta}_{\mathcal{G}} \Delta}{\Gamma \mid e \Leftrightarrow_R e'' : A \vdash^{\Theta}_{\mathcal{G}} \Delta} \; \mathit{transL}$$

Core conversion compatibility:

$$\frac{}{\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} x \Leftrightarrow_R x : A \mid \Delta} \; \mathit{VR} \qquad \frac{}{\Gamma \mid \alpha \Leftrightarrow_R x : A \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta} \; \mathit{VL}$$

$$\frac{c \Leftrightarrow_R c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\alpha.c \Leftrightarrow_R \mu\alpha.c' : A \mid \Delta} \; \mathit{AR} \qquad \frac{c \Leftrightarrow_R c' : \left(\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}x.c \Leftrightarrow_R \tilde{\mu}x.c' : A \vdash^{\Theta}_{\mathcal{G}} \Delta} \; \mathit{AL}$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v' : A \mid \Delta \quad \Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \Gamma \mid e \Leftrightarrow_R e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta}{\langle v \| e \rangle \Leftrightarrow_R \langle v' \| e' \rangle : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)} \; \mathit{Cut}$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v' : A \mid \Delta \quad \Theta \vdash_{\mathcal{G}} A =_{\beta\eta} B : \mathcal{S}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v' : B \mid \Delta} \; \mathit{TCR} \quad \frac{\Gamma \mid e \Leftrightarrow_R e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta \quad \Theta \vdash_{\mathcal{G}} A =_{\beta\eta} B : \mathcal{S}}{\Gamma \mid e \Leftrightarrow_R e' : B \vdash^{\Theta}_{\mathcal{G}} \Delta} \; \mathit{TCL}$$

FIGURE 7.1. Core parallel conversion rules.

(Co-)data conversion compatibility:

Given $\textbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\textbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y_i : l_i}} \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\theta = \left\{\overrightarrow{C/X}\right\} \;\; \overrightarrow{\Theta \vdash_{\mathcal{G}} C'_i\theta : l_i\theta} \;\; \theta' = \left\{\overrightarrow{C'_i/Y_i}\right\}\theta \;\; \overrightarrow{\Gamma \mid e \Leftrightarrow_R e' : B_{ij}\theta' \vdash^{\Theta}_{\mathcal{G}} \Delta}^{j} \;\; \overrightarrow{\Gamma \mid v \Leftrightarrow_R v' : A_{ij}\theta' \vdash^{\Theta}_{\mathcal{G}} \Delta}^{j}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{K}_i^{\overrightarrow{C'}}(\overrightarrow{e}, \overrightarrow{v}) \Leftrightarrow_R \mathsf{K}_i^{\overrightarrow{C'}}(\overrightarrow{e'}, \overrightarrow{v'}) : \mathsf{F}(\overrightarrow{C}) \mid \Delta}\; \mathsf{F}R_{\mathsf{K}_i}$$

$$\frac{\theta = \left\{\overrightarrow{C/X}\right\} \quad \overrightarrow{c_i \Leftrightarrow_R c'_i : \left(\Gamma, \overrightarrow{x_i : A_i\theta} \vdash^{\Theta, \overrightarrow{Y_i : l_i\theta}}_{\mathcal{G}} \overrightarrow{\alpha_i : B_i\theta}, \Delta\right)}^{i}}{\Gamma \mid \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y_i : l_i}}(\overrightarrow{\alpha_i}, \overrightarrow{x_i}).c_i}^{i}\right] \Leftrightarrow_R \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y_i : l_i}}(\overrightarrow{\alpha_i}, \overrightarrow{x_i}).c'_i}^{i}\right] : \mathsf{F}(\overrightarrow{C}) \vdash^{\Theta}_{\mathcal{G}} \Delta}\; \mathsf{F}L$$

Given $\textbf{codata}\,\mathsf{G}(\overrightarrow{X : k}) : \mathcal{S}\,\textbf{where}\,\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\overrightarrow{X}) \vdash^{\overrightarrow{Y_i : l_i}} \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\theta = \left\{\overrightarrow{C/X}\right\} \quad \overrightarrow{c_i \Leftrightarrow_R c'_i : \left(\Gamma, \overrightarrow{x_i : A_i\theta} \vdash^{\Theta, \overrightarrow{Y_i : l_i\theta}}_{\mathcal{G}} \overrightarrow{\alpha_i : B_i\theta}, \Delta\right)}^{i}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y_i : l_i}}[\overrightarrow{x_i}, \overrightarrow{\alpha_i}].c_i}^{i}\right) \Leftrightarrow_R \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y_i : l_i}}[\overrightarrow{x_i}, \overrightarrow{\alpha_i}].c'_i}^{i}\right) : \mathsf{G}(\overrightarrow{C}) \mid \Delta}\; \mathsf{G}R$$

$$\frac{\theta = \left\{\overrightarrow{C/X}\right\} \;\; \overrightarrow{\Theta \vdash_{\mathcal{G}} C'_i : l_i} \;\; \theta' = \left\{\overrightarrow{C'_i/Y_i}\right\}\theta \;\; \overrightarrow{\Gamma \mid v \Leftrightarrow_R v' : A_{ij}\theta' \vdash^{\Theta}_{\mathcal{G}} \Delta}^{j} \;\; \overrightarrow{\Gamma \mid e \Leftrightarrow_R e' : B_{ij}\theta' \vdash^{\Theta}_{\mathcal{G}} \Delta}^{j}}{\Gamma \mid \mathsf{O}_i^{\overrightarrow{C'_i}}[\overrightarrow{v}, \overrightarrow{e}] \Leftrightarrow_R \mathsf{O}_i^{\overrightarrow{C'_i}}[\overrightarrow{v'}, \overrightarrow{e'}] : \mathsf{G}(\overrightarrow{C}) \vdash^{\Theta}_{\mathcal{G}} \Delta}\; \mathsf{G}L_{\mathsf{O}_i}$$

FIGURE 7.2. Parallel conversion rules for (co-)data types.

b) $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta$ *is derivable if and only if* $\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v : A \mid \Delta$ *is.*

c) $\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ *is derivable if and only if* $\Gamma \mid e \Leftrightarrow_R e : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ *is.*

Proof. By (mutual) induction on the given typing derivations. □

Lemma 7.9.

a) $c =_R c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ *is derivable if and only if* $c \Leftrightarrow_R c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ *is.*

b) $\Gamma \vdash^{\Theta}_{\mathcal{G}} v =_R v' : A \mid \Delta$ *is derivable if and only if* $\Gamma \vdash^{\Theta}_{\mathcal{G}} v \Leftrightarrow_R v' : A \mid \Delta$ *is.*

c) $\Gamma \mid e =_R e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ *is derivable if and only if* $\Gamma \mid e \Leftrightarrow_R e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ *is.*

Proof. By (mutual) induction on the given typing derivations. □

To show adequacy of the typing derivations, we need to be assured that enough of the syntax of the language is well-behaved, in whatever sense of “well-behaved” that we have chosen. For input and output abstractions, this is straightforward to show by type expansion (Lemma 7.2) which came from the saturation condition for worlds.

Lemma 7.10 (Strong activation)**.** *Given a binary* $\mathbb{T}_{\mathcal{S}}$*-type* $\mathbb{A}$*,*

a) $\mu\alpha.c \;\mathbb{A}\; \mu\alpha.c'$ *if* $c\left\{E/\alpha\right\} \perp\!\!\!\perp c'\left\{E'/\alpha\right\}$ *for all* $E \;\mathbb{A}|_{\mathbb{V}}\; E'$*, and*

b) $\tilde{\mu}x.c \;\mathbb{A}\; \tilde{\mu}x.c'$ *if* $c\left\{V/x\right\} \perp\!\!\!\perp c'\left\{V'/x\right\}$ *for all* $V \;\mathbb{A}|_{\mathbb{V}}\; V'$*.*

Proof. a) Observe that for all $E \;\mathbb{A}|_{\mathbb{V}}\; E'$ we have

$$\langle(\mu\alpha.c, \mu\alpha.c')\|(E, E')\rangle = (\langle\mu\alpha.c\|E\rangle, \langle\mu\alpha.c'\|E'\rangle) \rightsquigarrow (c\left\{E/\alpha\right\}, c'\left\{E'/\alpha\right\}) \in \perp\!\!\!\perp$$

Therefore, $\mu\alpha.c \;\mathbb{A}\; \mu\alpha.c'$ by head expansion (Lemma 7.2).

b) Analogous to part 1 by duality. □

However, we also need to know that enough of the structures and case abstractions are also well-behaved. Therefore, we extend the idea that a substitution strategy is focalizing—that is, it contains enough structures and abstractions—to the worlds of the model. For any safety condition $\mathbb{S} = (\top\!\!\!\top, \mathbb{I}, \perp\!\!\!\perp)$, an $\mathbb{S}$-world $\mathbb{T}_{\mathcal{S}} = (\mathbb{U}_{\mathcal{S}}, \mathbb{V}_{\mathcal{S}}, \mathbb{W}_{\mathcal{S}})$ is *focalizing* whenever the following conditions hold:

- $\mathsf{K}^{\vec{C}}(\vec{E}, \vec{V})\ \mathbb{W}_{\mathcal{S}}\ \mathsf{K}^{\vec{C}}(\vec{E'}, \vec{V'})$ and $\mathsf{O}^{\vec{C}}[\vec{V}, \vec{E}]\ \mathbb{W}_{\mathcal{S}}\ \mathsf{O}^{\vec{C}}[\vec{V'}, \vec{E'}]$ if $\overrightarrow{V\ \mathbb{W}_{\mathcal{S}}\ V'}$ and $\overrightarrow{E\ \mathbb{W}_{\mathcal{S}}\ E'}$, and
- $\mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y{:}l}}[\vec{x}, \vec{\alpha}].c}\right) \mathbb{W}_{\mathcal{S}}\ \mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y{:}l}}[\vec{x}, \vec{\alpha}].c'}\right)$ and $\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{\alpha}, \vec{x}).c}\right] \mathbb{W}_{\mathcal{S}}\ \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{\alpha}, \vec{x}).c'}\right]$ such that $c\left\{\overrightarrow{C/X}, \overrightarrow{V/x}, \overrightarrow{E/\alpha}\right\} \perp\!\!\!\perp c'\left\{\overrightarrow{C/X}, \overrightarrow{V'/x}, \overrightarrow{E'/\alpha}\right\}$ for all $\overrightarrow{C\ [\![l]\!]\ \mathbb{C}}$ and $\overrightarrow{V\ \mathbb{W}_{\mathcal{S}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}\ V'}$ and $\overrightarrow{E\ \mathbb{W}_{\mathcal{S}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}\ E'}$.

From this assumption, we can show the adequacy of the typing rules.

Lemma 7.11 (Parallel conversion adequacy)**.** *For any focalizing strategies $\vec{\mathcal{S}}$, safety condition $\mathbb{S} = (\top\!\!\!\top, \mathbb{I}, \perp\!\!\!\perp)$ such that $\perp\!\!\!\perp$ is transitive, family of focalizing $\mathbb{S}$-worlds $\mathbb{T}_{\mathcal{S}}$, and size measurement,*

a) if $c \Leftrightarrow_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} c' : \left(\Gamma \vdash_{\mathcal{G}}^{\Theta} \Delta\right)$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $c \Leftrightarrow c' : \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)$,

b) if $\Gamma \vdash_{\mathcal{G}}^{\Theta} v \Leftrightarrow_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} v' : A \mid \Delta$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Gamma \vDash_{\mathcal{G}}^{\Theta} v \Leftrightarrow v' : A \mid \Delta$, and

c) if $\Gamma \mid e \Leftrightarrow_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} e' : A \vdash_{\mathcal{G}}^{\Theta} \Delta$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Gamma \mid e \Leftrightarrow e' : A \vDash_{\mathcal{G}}^{\Theta} \Delta$.

Proof. By mutual induction on the derivations for parallel conversion in Figures 7.1 and 7.2, demonstrating that each inference rule is sound similar to Lemma 7.7.

For the conversion rules, $[\![symm]\!]$ and $[\![trans]\!]$ are sound by the symmetry and transitivity assumed for the $\perp\!\!\!\perp$ relation. The $[\![Rw]\!]$ rule for commands follows because the $\rightsquigarrow$ computation relation in $\top\!\!\!\top$ allows for only one side or the other to reduce. For example, to show the soundness of $\mu_{\vec{\mathcal{S}}}$ rule:

$$\frac{c \Leftrightarrow c : \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right) \quad c \succ_{\mu_{\vec{\mathcal{S}}}} c' \quad c' \Leftrightarrow c' : \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)}{c \Leftrightarrow c' : \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)}\ [\![\succ]\!]$$

suppose that $\sigma \in \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)\mathbf{seq}$, so from the premises we know that $c\,\{\mathit{left}(\sigma)\} \perp\!\!\!\perp c\,\{\mathit{right}(\sigma)\}$ and $c'\,\{\mathit{left}(\sigma)\} \perp\!\!\!\perp c'\,\{\mathit{right}(\sigma)\}$. Thus, since

$$(c\,\{\mathit{left}(\sigma)\}\,, c'\,\{\mathit{right}(\sigma)\}) \rightsquigarrow (c'\,\{\mathit{left}(\sigma)\}\,, c'\,\{\mathit{right}(\sigma)\}) \in \perp\!\!\!\perp$$

we have $c\,\{\mathit{left}(\sigma)\} \perp\!\!\!\perp c\,\{\mathit{right}(\sigma)\}$ by $\mathbb{S}$'s closure under expansion. Similarly, we could expand the right-hand side instead of the left-hand side, which gives us the reverse rewriting rule $\prec$. Additionally, the $\succ R$ and $\succ L$ rules for terms and co-terms follows because every semantic $\mathbb{T}_{\mathcal{S}}$-type $\mathbb{A}$ is equal to $\mathbb{A}|_{\mathbb{V}_{\mathcal{S}}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$, and each rewriting rule on (co-)terms is simulated in the type by a rewriting rule on commands. For terms, we have the η_μ rule for terms of any type, so to show that

$$\frac{\Gamma \vDash_{\mathcal{G}}^{\Theta} \mu\alpha.\, \langle v \| \alpha\rangle \Leftrightarrow \mu\alpha.\, \langle v \| \alpha\rangle : A \mid \Delta \quad \mu\alpha.\, \langle v \| \alpha\rangle \succ_{\eta_\mu} v \quad \Gamma \vDash_{\mathcal{G}}^{\Theta} v \Leftrightarrow v : A \mid \Delta}{\Gamma \vDash_{\mathcal{G}}^{\Theta} \mu\alpha.\, \langle v \| \alpha\rangle \Leftrightarrow v : A \mid \Delta} [\![\succ R]\!]$$

suppose that $\sigma \in \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)\mathbf{seq}$ and $[\![A]\!]_{sem(\sigma)} \in SemType(\mathbb{T}_{\mathcal{S}})$ for some $\mathcal{S}$, so from the premises we know that $\mu\alpha.\, \langle v\|\alpha\rangle \{left(\sigma)\}\ [\![A]\!]_{sem(\sigma)}\ \mu\alpha.\, \langle v\|\alpha\rangle \{right(\sigma)\}$ and $v\{left(\sigma)\}\ [\![A]\!]_{sem(\sigma)}\ v\{right(\sigma)\}$. Letting $E\ [\![A]\!]_{sem(\sigma)}\ E'$, note that we have the step

$$(\langle \mu\alpha.\, \langle v\|\alpha\rangle \{left(\sigma)\} \| E\rangle, \langle v\{right(\sigma)\} \| E'\rangle) \rightsquigarrow (\langle v\{left(\sigma)\} \| E\rangle, \langle v\{left(\sigma)\} \| E'\rangle) \in \perp\!\!\!\perp$$

and so $\langle \mu\alpha.\, \langle v\|\alpha\rangle \| E\rangle \perp\!\!\!\perp \langle v \| E'\rangle$ by $\mathbb{S}$'s closure under expansion. Similarly, the $\eta^{\mathcal{G}}$ rewriting rule is justified individually for each (co-)data type because case abstractions are (co-)values (by the focalization assumption on each strategy) by inspecting the positive or negative definitions of the type. In particular, the same justification as η_μ holds for values of a known co-data type defined as $Neg(\mathbb{A}_{obs})$, where we only need to check that $\langle V \| E\rangle \succ_\beta \langle V' \| E\rangle$ for each $E \in \mathbb{A}_{obs}$ to conclude that $V\ \mathbb{A}_{obs}|_{\mathbb{V}_{\mathcal{S}-}}^{\mathbb{W}_{\mathcal{S}+}}\ V'$, so that $V\ Neg(\mathbb{A}_{obs})\ V'$ by restricted double orthogonal introduction (Property 7.5 (c)). The soundness of $\prec R$ follows similarly because either side of the relation may reduce while the other stays the same, and the soundness of $\succ L$ and $\prec L$ follow analogously by duality.

We now move on to demonstrating the soundness of the many compatibility rules. The interpretation of the core compatibility typing rules is sound as follows:

- To show $\dfrac{\Gamma \vDash_{\mathcal{G}}^{\Theta} v \Leftrightarrow v' : A \mid \Delta \quad \Theta \vDash_{\mathcal{G}} A : \mathcal{S} \quad \Gamma \mid e \Leftrightarrow e' : A \vDash_{\mathcal{G}}^{\Theta} \Delta}{\langle v \| e\rangle \Leftrightarrow \langle v' \| e'\rangle : \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)} [\![Cut]\!]$ suppose that $\sigma \in \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)\mathbf{seq}$, so we know that $A\{left(\sigma)\}\ [\![\mathcal{S}]\!]_{sem(\sigma)}\ [\![A]\!]_{sem(\sigma)}$, $v\{left(\sigma)\}\ [\![A]\!]_{sem(\sigma)}\ v'\{right(\sigma)\}$, and $e\{left(\sigma)\}\ [\![A]\!]_{sem(\sigma)}\ e'\{right(\sigma)\}$from the premises. Note that by definition of $[\![\mathcal{S}]\!]_{sem(\sigma)}$, $[\![A]\!]_{sem(\sigma)}$ must be a $\mathbb{T}_{\mathcal{S}}$-type, so that $[\![A]\!]_{sem(\sigma)} = [\![A]\!]_{sem(\sigma)}\Big|_{\mathbb{V}_{\mathcal{S}}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}} = [\![A]\!]_{sem(\sigma)}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$. Therefore, since $\langle (v, v') \| (e, e')\rangle = (\langle v \| e\rangle, \langle v' \| e'\rangle) \in \top\!\!\!\top$ we have that $\langle v \| e\rangle \perp\!\!\!\perp \langle v' \| e'\rangle$.

- To show $\dfrac{}{\Gamma, x : A \vDash^{\Theta}_{\mathcal{G}} x \Leftrightarrow x : A \mid \Delta} \llbracket VR \rrbracket$ suppose that $\sigma \in \left(\Gamma, x : A \vDash^{\Theta}_{\mathcal{G}} \Delta\right) \mathbf{seq}$, so that $(x\,\{\mathit{left}(\sigma)\}, x\,\{\mathit{right}(\sigma)\}) = (\mathit{left}(\sigma)(x), \mathit{right}(\sigma)(x)) = \sigma(x) \in \llbracket A \rrbracket_{\mathit{sem}(\sigma)}$.
- The soundness of $\dfrac{c \Leftrightarrow c' : \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)}{\Gamma \vDash^{\Theta}_{\mathcal{G}} \mu\alpha.c \Leftrightarrow \mu\alpha.c' : A \mid \Delta} \llbracket AR \rrbracket$ follows immediately from the premise and Lemma 7.10.
- To show $\dfrac{\Gamma \vDash^{\Theta}_{\mathcal{G}} v \Leftrightarrow v' : A \mid \Delta \quad \Theta \vDash_{\mathcal{G}} A =_{\beta\eta} B : \mathcal{S}}{\Gamma \vDash^{\Theta}_{\mathcal{G}} v \Leftrightarrow v' : B \mid \Delta} \llbracket TCR \rrbracket$ suppose that $\sigma \in \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right) \mathbf{seq}$ and $\llbracket A \rrbracket_{\mathit{sem}(\sigma)} \in \mathit{SemType}(\mathbb{T}_{\mathcal{S}})$ for some $\mathcal{S}$, so by the premise we know that $v\,\{\mathit{left}(\sigma)\}\ \llbracket A \rrbracket_{\mathit{sem}(\sigma)}\ v'\,\{\mathit{right}(\sigma)\}$ and $\llbracket A \rrbracket_{\mathit{sem}(\sigma)} = \llbracket B \rrbracket_{\mathit{sem}(\sigma)}$, therefore $v\,\{\mathit{left}(\sigma)\}\ \llbracket B \rrbracket_{\mathit{sem}(\sigma)}\ v'\,\{\mathit{right}(\sigma)\}$ as well.
- The soundness of $\llbracket VL \rrbracket$, $\llbracket AL \rrbracket$, and $\llbracket TCL \rrbracket$ follows dually to the cases for $\llbracket VR \rrbracket$, $\llbracket AR \rrbracket$, and $\llbracket TCR \rrbracket$.

Given a higher-order data type declaration (which subsumes simple data declarations),

$$\mathbf{data}\,\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y_i : l_i}} \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$$

the associated convertibility compatibility rules are sound as follows:

- To show

$$\dfrac{\overrightarrow{\Theta \vDash_{\mathcal{G}} C'_i : l_i} \quad \overrightarrow{\Gamma \mid e_j \Leftrightarrow e'_j : B_{ij}\theta \vDash^{\Theta}_{\mathcal{G}} \Delta}^{j} \quad \overrightarrow{\Gamma \mid v_j \Leftrightarrow v'_j : A_{ij}\theta \vDash^{\Theta}_{\mathcal{G}} \Delta}^{j}}{\Gamma \vDash^{\Theta}_{\mathcal{G}} \mathsf{K}_i^{\overrightarrow{C'}}(\overrightarrow{e_j}^{j}, \overrightarrow{v_j}^{j}) \Leftrightarrow \mathsf{K}_i^{\overrightarrow{C'}}(\overrightarrow{e'_j}^{j}, \overrightarrow{v'_j}^{j}) : \mathsf{F}(\overrightarrow{C}) \mid \Delta} \llbracket \mathsf{F}R_{\mathsf{K}_i} \rrbracket$$

where $\theta' = \left\{\overrightarrow{C'_i/Y_i}\right\}\theta$ and $\theta = \left\{\overrightarrow{C/X}\right\}$, suppose $\sigma \in \left(\overrightarrow{\Gamma_j}^{j}, \overrightarrow{\Gamma'_j}^{j} \vDash^{\Theta}_{\mathcal{G}} \overrightarrow{\Delta_j}^{j}, \overrightarrow{\Delta'_j}^{j}\right) \mathbf{seq}$ and $\mathsf{F}(\overrightarrow{C})\,\{\mathit{left}(\sigma)\}\ \llbracket \mathcal{S} \rrbracket_{\mathit{sem}(\sigma)}\ \left\llbracket \mathsf{F}(\overrightarrow{C}) \right\rrbracket_{\mathit{sem}(\sigma)}$ so we know that

$$C_i\,\{\mathit{left}(\sigma)\}\ \llbracket k_i \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket C_i \rrbracket_{\mathit{sem}(\sigma)}$$
$$C'_i\theta\,\{\mathit{left}(\sigma)\}\ \llbracket l_i \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket C'_i\theta \rrbracket_{\mathit{sem}(\sigma)}$$
$$A_{ij}\theta\,\{\mathit{left}(\sigma)\}\ \llbracket \mathcal{T}_{ij} \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket A_{ij}\theta \rrbracket_{\mathit{sem}(\sigma)}$$
$$B_{ij}\theta\,\{\mathit{left}(\sigma)\}\ \llbracket \mathcal{R}_{ij} \rrbracket_{\mathit{sem}(\sigma)}\ \llbracket B_{ij}\theta \rrbracket_{\mathit{sem}(\sigma)}$$

$$\overrightarrow{e_j}^j \,\{\mathit{left}(\sigma)\}\; [\![B_{ij}\theta]\!]\; e'_j \,\{\mathit{right}(\sigma)\}$$
$$v_j \,\{\mathit{left}(\sigma)\}\; [\![A_{ij}\theta]\!]\; v'_j \,\{\mathit{right}(\sigma)\}$$

by the premises. We now proceed by induction on the number of (co-)values among e_j, e'_j, v_j, v'_j:

- ∗ If e_j, e'_j, v_j, v'_j are all (co-)values, then both the constructions $\mathsf{K}_i^{\vec{C'}}(\overrightarrow{e_j}^j, \overrightarrow{v_j}^j)$ and $\mathsf{K}_i^{\vec{C'}}(\overrightarrow{e'_j}^j, \overrightarrow{v'_j}^j)$ are values because $\mathcal{S}$ is focalizing. It then follows that both constructions are in $\mathbb{W}_{\mathcal{S}}|_{\mathbb{V}_{\mathcal{S}}}$ since $\mathbb{T}_{\mathcal{S}}$ is focalizing, so they are related by

  $$\bigcup_i \left[\!\!\left[\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^j \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^j\right)\right]\!\!\right]_{\mathit{sem}(\sigma),\overrightarrow{(C,[\![C]\!]_{\mathit{sem}(\sigma)})/X}}$$

  Thus, both constructions are also related by

  $$\begin{aligned}&\left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathit{sem}(\sigma)}\\ &= \mathit{Pos}\left(\bigcup_i \left[\!\!\left[\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^j \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{F}(\vec{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^j\right)\right]\!\!\right]_{\mathit{sem}(\sigma),\overrightarrow{(C,[\![C]\!]_{\mathit{sem}(\sigma)})/X}}\right)\end{aligned}$$

  by restricted double orthogonal introduction (Property 7.5 (c)).

- ∗ If $\overrightarrow{e_j}^j = \vec{E}, e_j, \vec{e'}$ where $e_j \notin \mathit{CoValue}_{\mathcal{R}_{ij}}$, then observe that for all $E' \in \left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathit{sem}(\sigma)}$ we have the step

  $$\begin{aligned}&\left\langle \mathsf{K}_i^{\vec{C'}}(\vec{E}, e_j, \vec{e'}, \overrightarrow{v_j}^j) \middle\| E' \right\rangle\\ &\mapsto_{\varsigma} \left\langle \mu\alpha. \left\langle \mu\beta_j. \left\langle \mathsf{K}_i^{\vec{C'}}(\vec{E}, \beta_j, \vec{e'}, \overrightarrow{v_j}^j) \middle\| \alpha \right\rangle \middle\| e_j \right\rangle \middle\| E' \right\rangle \in \perp\!\!\!\perp\end{aligned}$$

  which lands in $\perp\!\!\!\perp$ by the inductive hypothesis and Lemma 7.10 applied to the $\mathbb{T}_{\mathcal{S}}$-type $\left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathit{sem}(\sigma)}$ and to the $\mathbb{T}_{\mathcal{R}_{ij}}$-type $[\![B_{ij}\theta]\!]_{\mathit{sem}(\sigma)}$. Thus, $\mathsf{K}_i^{\vec{C'}}(\vec{E}, e_j, \vec{e'}, \overrightarrow{v_j}^j) \in \left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathit{sem}(\sigma)}$ by Lemma 7.2.

  - $*$ If all of $\overrightarrow{e_j}^j = \overrightarrow{E_j}^j$ and $\overrightarrow{v_j}^j = \vec{V}, v_j, \overrightarrow{v'}$ where $v_j \notin \mathit{CoValue}_{\mathcal{T}_{ij}}$, then observe that for all $E' \in \left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathit{sem}(\sigma)}$ we have the step

$$\begin{aligned}&\left\langle \mathsf{K}_i^{\overrightarrow{C'}}(\overrightarrow{E_j}^j, \vec{V}, v_j, \overrightarrow{v'}) \middle\| E' \right\rangle \\ &\mapsto_{\varsigma} \left\langle \mu\alpha. \left\langle v_j \middle\| \mu y_j. \left\langle \mathsf{K}_i^{\overrightarrow{C'}}(\overrightarrow{}^j E_j, \vec{V}, y_j, \overrightarrow{v'}) \middle\| \alpha \right\rangle \right\rangle \middle\| E' \right\rangle \in \perp\!\!\!\perp\end{aligned}$$

    which lands in $\perp\!\!\!\perp$ by the inductive hypothesis and Lemma 7.10 applied to the $\mathbb{T}_{\mathcal{S}}$-type $\left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathit{sem}(\sigma)}$ and to the $\mathbb{T}_{\mathcal{T}_{ij}}$-type $[\![A_{ij}\theta]\!]_{\mathit{sem}(\sigma)}$. Thus, $\mathsf{K}_i^{\overrightarrow{C'}}(\overrightarrow{E_j}^j, \vec{V}, v_j, \overrightarrow{v'}) \in \left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathit{sem}(\sigma)}$ by Lemma 7.2.

  - $*$ The cases where we have a non-(co-)value among $\overrightarrow{e'_j}^j$ or $\overrightarrow{v'_j}^j$ are analogous to the previous two cases.

- To show

$$\frac{\overrightarrow{c_i \Leftrightarrow c'_i : \left(\Gamma, \overrightarrow{x_i : A_i\{\overrightarrow{C/X}\}} \vDash_{\mathcal{G}}^{\Theta, \overrightarrow{Y_i : l_i}} \overrightarrow{\alpha_i : B_i\{\overrightarrow{C/X}\}}, \Delta\right)}^i}{\Gamma \mid \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y_i : l_i}}(\vec{\alpha_i}, \vec{x_i}).c_i}^i\right] \Leftrightarrow \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y_i : l_i}}(\vec{\alpha_i}, \vec{x_i}).c'_i}^i\right] : \mathsf{F}(\vec{C}) \vDash_{\mathcal{G}}^{\Theta} \Delta} [\![\mathsf{F}L]\!]$$

  suppose that $\sigma \in \left(\Gamma \vDash_{\mathcal{G}}^{\Theta} \Delta\right)\mathbf{seq}$ and $\mathsf{F}(\vec{C})\,\{\mathit{left}(\sigma)\}\ [\![\mathcal{S}]\!]_{\mathit{sem}(\sigma)}\ \left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathit{sem}(\sigma)}$ so we know that

$$\begin{gathered}C_i\,\{\mathit{left}(\sigma)\}\ [\![l_i]\!]_{\mathit{sem}(\sigma)}\ [\![C_i]\!]_{\mathit{sem}(\sigma)} \\ A_{ij}\theta\,\{\mathit{left}(\sigma)\}\ [\![\mathcal{T}_{ij}]\!]_{\mathit{sem}(\sigma)}\ [\![A_{ij}\theta]\!]_{\mathit{sem}(\sigma)} \\ B_{ij}\theta\,\{\mathit{left}(\sigma)\}\ [\![\mathcal{R}_{ij}]\!]_{\mathit{sem}(\sigma)}\ [\![B_{ij}\theta]\!]_{\mathit{sem}(\sigma)}\end{gathered}$$

  and for all

$$\begin{gathered}C'_{ij}\ [\![l_{ij}]\!]_{\mathit{sem}(\sigma)}\ \mathbb{C}'_{ij} \\ E_{ij}\ \left[\!\!\left[B_{ij}\left\{\overrightarrow{C/X}\right\}\right]\!\!\right]_{\mathit{sem}(\sigma), \overrightarrow{\mathbb{C}'_{ij}/Y_{ij}}}\ E'_{ij} \\ V_{ij}\ \left[\!\!\left[A_{ij}\left\{\overrightarrow{C/X}\right\}\right]\!\!\right]_{\mathit{sem}(\sigma), \overrightarrow{\mathbb{C}'_{ij}/Y_{ij}}}\ V'_{ij}\end{gathered}$$

we have $c\,\{\mathit{left}(\sigma)'\} \perp\!\!\!\perp c'\,\{\mathit{right}(\sigma)\}$ where

$$\sigma' = \sigma\overrightarrow{(C,\mathbb{C})/X}, \overrightarrow{(C'_{ij},\mathbb{C}'_{ij})/Y_{ij}}^{ij}, \overrightarrow{(V_{ij},V'_{ij})/x_{ij}}^{ij}, \overrightarrow{(E_{ij},E'_{ij})/\alpha_{ij}}^{ij}$$

We know that $\mathbb{W}_{\mathcal{T}_{ij}}\Big|_{\mathbb{V}_{\mathcal{T}_{ij}}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{T}_{ij}}}} \sqsubseteq \left\llbracket A_{ij}\left\{\overrightarrow{C/X}\right\}\right\rrbracket_{sem(\sigma),\overrightarrow{\mathbb{C}'_{ij}/Y_{ij}}}$ and $\mathbb{W}_{\mathcal{R}_{ij}}\Big|_{\mathbb{V}_{\mathcal{R}_{ij}}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{R}_{ij}}}} \sqsubseteq \left\llbracket B_{ij}\left\{\overrightarrow{C/X}\right\}\right\rrbracket_{sem(\sigma),\overrightarrow{\mathbb{C}'_{ij}/Y_{ij}}}$ by Lemma 7.1, so the two case abstractions must be related by $\mathbb{W}_{\mathcal{S}}$ since $\mathbb{T}_{\mathcal{S}}$ is focalizing. Furthermore, for each

$$(V,V') \in \bigcup_i \left\llbracket \mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)\right\rrbracket_{sem(\sigma),\overrightarrow{(C,\llbracket C\rrbracket_{sem(\sigma)})/X}}$$

we have a β step

$$\left(\left\langle \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y_i{:}l_i}}(\overrightarrow{\alpha_i},\overrightarrow{x_i}).c_i}^{i}\right]\{\mathit{left}(\sigma)\}\middle\| V\right\rangle, \left\langle \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y_i{:}l_i}}(\overrightarrow{\alpha_i},\overrightarrow{x_i}).c'_i}^{i}\right]\{\mathit{right}(\sigma)\}\middle\| V'\right\rangle\right)$$
$$\rightsquigarrow (c,c') \in \perp\!\!\!\perp$$

Thus, by the safety condition $\mathcal{S}$'s closure under expansion, we have that the two case abstractions are related by

$$\left(\bigcup_i \left\llbracket \mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)\right\rrbracket_{sem(\sigma),\overrightarrow{(C,\llbracket C\rrbracket_{sem(\sigma)})/X}}\right)^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}-}}$$

which means they are also related by

$$\left\llbracket \mathsf{F}(\overrightarrow{C})\right\rrbracket_{sem(\sigma)} = \mathit{Pos}\left(\bigcup_i \left\llbracket \mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y_i{:}l_i}} \mathsf{F}(\overrightarrow{X}) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)\right\rrbracket_{sem(\sigma),\overrightarrow{(C,\llbracket C\rrbracket_{sem(\sigma)})/X}}\right)$$

by restricted double orthogonal introduction Property 7.5 (c).

Note that the logical rules for (both primitive and noetherian) recursive data types are the same as those for higher-order data types, so their soundness follows similarly.

The recursive rules for $\exists_{\mathsf{Ix}}$ and $\exists_{<\mathsf{Ord}}$ are sound as follows:

- To show

$$\frac{c_0 \Leftrightarrow c'_0 : \left(\Gamma, x : A\ 0 \vDash_{\mathcal{G}}^{\Theta} \Delta\right) \quad c_1 \Leftrightarrow c'_1 : \left(\Gamma, x : A\ (j{+}1) \vDash_{\mathcal{G}}^{\Theta, j:\mathsf{Ix}} \alpha : A\ j, \Delta\right)}{\Gamma \mid \tilde{\mu}[0{:}\mathsf{Ix} \mathbin{@} x.c_0 \mid j{+}1{:}\mathsf{Ix} \mathbin{@_\alpha} x.c_1] \Leftrightarrow \tilde{\mu}[0{:}\mathsf{Ix} \mathbin{@} x.c'_0 \mid j{+}1{:}\mathsf{Ix} \mathbin{@_\alpha} x.c'_1] : \exists_{\mathsf{Ix}}(A) \vDash_{\mathcal{G}}^{\Theta} \Delta}\ \exists_{\mathsf{Ix}} L_{rec}$$

suppose that $\sigma \in \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right) \mathbf{seq}$ and that $\exists_{\mathsf{lx}}(A)\,\{\mathit{left}(\sigma)\}\ \ [\![\mathcal{S}]\!]_{\mathit{sem}(\sigma)}\ \ [\![\exists_{\mathsf{lx}}(A)]\!]_{\mathit{sem}(\sigma)}$, so it must be that $A\,\{\mathit{left}(\sigma)\}\ [\![\mathcal{S}]\!]_{\mathit{sem}(\sigma)}\ [\![A]\!]_{\mathit{sem}(\sigma)}$ and thus from the premises we know that

$$c_0\,\{\mathit{left}(\sigma), V_0/x\} \perp\!\!\!\perp c_0'\,\{\mathit{right}(\sigma), V_0'/x\}$$
$$c_1\,\{\mathit{left}(\sigma), V_1/x, E/\alpha\} \perp\!\!\!\perp c_1'\,\{\mathit{right}(\sigma), V_1'/x, E'/\alpha\}$$

for all $V_0\ \ [\![A]\!]_{\mathit{sem}(\sigma)}(0)\ \ V_0'$, $M\ \ [\![\mathsf{lx}]\!]_{\mathit{sem}(\sigma)}\ \ \mathbb{M}$, $V_1\ \ [\![A]\!]_{\mathit{sem}(\sigma)}(+1(\mathbb{M}))\ \ V_1'$, and $E\ [\![A]\!]_{\mathit{sem}(\sigma)}(\mathbb{M})\ E'$. Furthermore, since

$$[\![\exists_{\mathsf{lx}}(A)]\!]_{\mathit{sem}(\sigma)} = \mathit{Neg}\left(\left[\!\!\left[\_ \mathbin{@} \_ : \left(A\ j \vdash^{j:\mathsf{lx}} \exists_{\mathsf{lx}}(A) \mid\ \right)\right]\!\!\right]_{\mathit{sem}(\sigma)}\right)$$

, we can then proceed by to show that for every

$$V \left[\!\!\left[\_ \mathbin{@} \_ : \left(A\ j \vdash^{j:\mathsf{lx}} \exists_{\mathsf{lx}}(A) \mid\ \right)\right]\!\!\right]_{\mathit{sem}(\sigma)} V'$$

we step to a pair of commands in $\perp\!\!\!\perp$ by induction on $M\ \sim^{\mathbb{N}}$ underlying $M[\![\mathsf{lx}]\!]_{\mathit{sem}(\sigma)}\mathbb{M}$:

∗ in the case of

$$(0 \mathbin{@} V_0) \left[\!\!\left[\_ \mathbin{@} \_ : \left(A\ j \vdash^{j:\mathsf{lx}} \exists_{\mathsf{lx}}(A) \mid\ \right)\right]\!\!\right]_{\mathit{sem}(\sigma)} (0 \mathbin{@} V_0')$$

where $V_0\ [\![A]\!]_{\mathit{sem}(\sigma)}(0)\ V_0'$, note that we have the β step

$$\begin{aligned}
&\begin{pmatrix}
\langle 0 \mathbin{@} V_0 \| \tilde{\mu}[0{:}\mathsf{lx} \mathbin{@} x.c_0 \mid j{+}1{:}\mathsf{lx} \mathbin{@_\alpha} x.c_1]\,\{\mathit{left}(\sigma)\}\rangle\,, \\
\langle 0 \mathbin{@} V_0 \| \tilde{\mu}[0{:}\mathsf{lx} \mathbin{@} x.c_0' \mid j{+}1{:}\mathsf{lx} \mathbin{@_\alpha} x.c_1']\,\{\mathit{right}(\sigma)\}\rangle
\end{pmatrix} \\
&\rightsquigarrow (c_0\,\{\mathit{left}(\sigma), V_0/x\}\,, c_0'\,\{\mathit{right}(\sigma), V_0'/x\}) \in \perp\!\!\!\perp
\end{aligned}$$

Thus, by the safety conditions $\mathbb{S}$'s closure under expansion, we have that the two case abstractions are related by

$$\left[\!\!\left[\_ \mathbin{@} \_ : \left(A\ j \vdash^{j:\mathsf{lx}} \exists_{\mathsf{lx}}(A) \mid\ \right)\right]\!\!\right]^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}_{\mathit{sem}(\sigma)}$$

and so they are also related by $[\![\exists_{\mathsf{Ix}}(A)]\!]_{\mathit{sem}(\sigma)}$ by restricted double orthogonal introduction (Property 7.5 (c)).

- in the case of

$$(M{+}1 \mathbin{@} V_1) \left[\!\!\left[\_ \mathbin{@} \_ : \left(A \ j \vdash^{j:\mathsf{Ix}} \exists_{\mathsf{Ix}}(A) \mid \right)\right]\!\!\right]_{\mathit{sem}(\sigma)} (M{+}1 \mathbin{@} V_1')$$

where $M{+}1 \ [\![\mathsf{Ix}]\!]_{\mathit{sem}(\sigma)} \ {+1}(\mathbb{M})$ and $V_1 \ [\![A]\!]_{\mathit{sem}(\sigma)}(0) \ V_1'$, note that we have the β step

$$\begin{aligned}
&\begin{pmatrix} \langle M{+}1 \mathbin{@} V_1 \| \tilde{\mu}[0{:}\mathsf{Ix} \mathbin{@} x.c_0 \mid j{+}1{:}\mathsf{Ix} \mathbin{@_\alpha} x.c_1] \{\mathit{left}(\sigma)\}\rangle \,, \\ \langle M{+}1 \mathbin{@} V_1 \| \tilde{\mu}[0{:}\mathsf{Ix} \mathbin{@} x.c_0' \mid j{+}1{:}\mathsf{Ix} \mathbin{@_\alpha} x.c_1'] \{\mathit{right}(\sigma)\}\rangle \end{pmatrix} \\
&\rightsquigarrow \begin{pmatrix} \langle \mu\alpha.c_1 \{\mathit{left}(\sigma), M/j, V_1/x\} \| \tilde{\mu}[0{:}\mathsf{Ix} \mathbin{@} x.c_0 \mid j{+}1{:}\mathsf{Ix} \mathbin{@_\alpha} x.c_1] \{\mathit{left}(\sigma)\}\rangle \,, \\ \langle \mu\alpha.c_1' \{\mathit{right}(\sigma), M/j, V_1'/x\} \| \tilde{\mu}[0{:}\mathsf{Ix} \mathbin{@} x.c_0' \mid j{+}1{:}\mathsf{Ix} \mathbin{@_\alpha} x.c_1'] \{\mathit{right}(\sigma)\}\rangle \end{pmatrix} \\
&\in \perp\!\!\!\perp
\end{aligned}$$

where the result is related in $\perp\!\!\!\perp$ by the inductive hypothesis. Thus, by the safety conditions $\mathbb{S}$'s closure under expansion, we have that the two case abstractions are related by

$$\left[\!\!\left[\_ \mathbin{@} \_ : \left(A \ j \vdash^{j:\mathsf{Ix}} \exists_{\mathsf{Ix}}(A) \mid \right)\right]\!\!\right]^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}_{\mathit{sem}(\sigma)}$$

and so they are also related by $[\![\exists_{\mathsf{Ix}}(A)]\!]_{\mathit{sem}(\sigma)}$ by restricted double orthogonal introduction (Property 7.5 (c)).

– To show

$$\frac{c \Leftrightarrow c' : \left(\Gamma, x : A \ j \vdash^{\Theta, j<N}_{\mathcal{G}} \alpha : \exists_{<\mathsf{Ord}}(j, A), \Delta\right)}{\Gamma \mid \tilde{\mu}[j < N \mathbin{@_\alpha} x.c] \Leftrightarrow \tilde{\mu}[j < N \mathbin{@_\alpha} x.c'] : \exists_{<\mathsf{Ord}}(N, A) \vdash^{\Theta}_{\mathcal{G}} \Delta} \ \exists_{<\mathsf{Ord}} L_{rec}$$

suppose that $\sigma \in \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right) \mathbf{seq}$ and that $\exists_{<\mathsf{Ord}}(N, A) \{\mathit{left}(\sigma)\} \ [\![\mathcal{S}]\!]_{\mathit{sem}(\sigma)} \ [\![\exists_{<\mathsf{Ord}}(N, A)]\!]_{\mathit{sem}(\sigma)}$, so it must be that $N \{\mathit{left}(\sigma)\} \ [\![\mathsf{Ord}]\!] \ [\mathit{sem}(\sigma)][\![N]\!]_{\mathit{sem}(\sigma)}$ and $A \{\mathit{left}(\sigma)\} \ [\![\mathcal{S}]\!]_{\mathit{sem}(\sigma)} \ [\![A]\!]_{\mathit{sem}(\sigma)}$ and thus from the premises we know that

$$c \{\mathit{left}(\sigma), V/x, E/\alpha\} \perp\!\!\!\perp c' \{\mathit{right}(\sigma), V'/x, E'/\alpha\}$$

for all $M \; [\![< N]\!]_{sem(\sigma)} \; \mathbb{M}$, $V \; [\![A]\!]_{sem(\sigma)}(\mathbb{M}) \; V'$, and $E \; [\![\exists_{<\mathsf{Ord}}(j, A)]\!]_{sem(\sigma),\mathbb{M}/j} \; E'$. We now proceed to show that the two case abstractions step to a pair of related commands in $⫫$ when cut with any values related by $[\![\exists_{<\mathsf{Ord}}(j, A)]\!]_{sem(\sigma),\mathbb{M}/j}$ by noetherian induction on $M \; [\![\mathsf{Ord}]\!]_{sem(\sigma)} \; \mathbb{M}$. In particular, note that for any $V \; [\![\exists_{<\mathsf{Ord}}(N, A)]\!]_{sem(\sigma)} \; V'$ there is the following unrolling ν step:

$$
\begin{aligned}
&(\langle V \| \tilde{\mu}[j < N \mathbin{@_\alpha} x.c] \{\mathit{left}(\sigma)\}\rangle , \langle V' \| \tilde{\mu}[j < N \mathbin{@_\alpha} x.c'] \{\mathit{right}(\sigma)\}\rangle) \\
&\rightsquigarrow \begin{pmatrix} \langle V \| \tilde{\mu}[i < N \mathbin{@} x.c \{\mathit{left}(\sigma), \tilde{\mu}[j < i \mathbin{@_\alpha} x.c] \{\mathit{left}(\sigma)\}/\alpha\}]\rangle , \\ \langle V' \| \tilde{\mu}[i < N \mathbin{@} x.c' \{\mathit{right}(\sigma), \tilde{\mu}[j < i \mathbin{@_\alpha} x.c'] \{\mathit{right}(\sigma)\}/\alpha\}]\rangle \end{pmatrix} \in ⫫
\end{aligned}
$$

where the result is related in $⫫$ because

$$
\tilde{\mu}[j < i \mathbin{@_\alpha} x.c] \{\mathit{left}(\sigma), M'/j\} \; [\![\exists_{<\mathsf{Ord}}(j, A)]\!]_{sem(\sigma),\mathbb{M}'/j} \; \tilde{\mu}[j < i \mathbin{@_\alpha} x.c'] \{\mathit{right}(\sigma), M'/j\}
$$

for all $M' \; [\![< N]\!]_{sem(\sigma)} \; \mathbb{M}'$ by the inductive hypothesis. Thus, the two case abstractions are related in $[\![\exists_{<\mathsf{Ord}}(N, A)]\!]_{sem(\sigma)}$ by Lemma 7.2.

The soundness of the conversion compatibility rules for higher-order, primitive recursive, noetherian recursive co-data types follows analogously by duality, as do the recursion rules for universal quantifiers over Ix and Ord. □

Lemma 7.12 (Type adequacy)**.** *For any focalizing strategies $\vec{\mathcal{S}}$, safety condition $\mathbb{S} = (⫪, ⫫\!\!\!\!\!\!=, ⫫)$ such that $⫫$ is transitive, family of focalizing $\mathbb{S}$-worlds $\mathbb{T}_{\mathcal{S}}$, and size measurement,*

a) if $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $c : \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right)$,

b) if $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Gamma \vDash^{\Theta}_{\mathcal{G}} v : A \mid \Delta$, and

c) if $\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Gamma \mid e : A \vDash^{\Theta}_{\mathcal{G}} \Delta$.

d) if $c =_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} c' : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $c \Leftrightarrow c' : \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right)$,

e) if $\Gamma \vdash^{\Theta}_{\mathcal{G}} v =_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} v' : A \mid \Delta$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Gamma \vDash^{\Theta}_{\mathcal{G}} v \Leftrightarrow v' : A \mid \Delta$, and

f) if $\Gamma \mid e =_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then $\Gamma \mid e \Leftrightarrow e' : A \vDash^{\Theta}_{\mathcal{G}} \Delta$.

Proof. By Lemmas 7.8, 7.9 and 7.11, noting that $c \Leftrightarrow c \; : \; \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right)$ which is definitionally the same as $c : \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right)$, and similarly for (co-)terms. □

Finally, we demonstrate the adequacy of sequents, by finding an appropriate substitution for any well-formed sequent quantifying only over declarations and type variables. Finding a substitution for sequents that quantify over (co-)variables is more dependent on the instantiation of the model, but can be done in certain circumstances.

Lemma 7.13 (Sequent adequacy)**.** *For any focalizing strategies $\vec{\mathcal{S}}$, safety condition $\mathbb{S}$ and family of $\mathbb{S}$-worlds $\mathbb{T}_{\mathcal{S}}$, if $\left(\vdash^{\Theta}_{\mathcal{G}}\right)\mathbf{seq}$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then there is a size measurement such that there exists a $\sigma \in \left(\vDash^{\Theta}_{\mathcal{G}}\right)\mathbf{seq}$. Furthermore, given $\mathbb{W}_{\mathcal{S}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}} \sqsupset (\emptyset, \emptyset)$ for all $\mathcal{S}$, then if $\left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then there is a size measurement such that there exists a $\sigma \in \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}$.*

Proof. By induction on the derivation of $\left(\vDash^{\Theta}_{\mathcal{G}}\right)\mathbf{seq}$. Note that we can always choose a big enough size measurement by including a finite number of extra ordinals in $\mathbb{O}$ less than the chosen 0 equal to the length of Θ, which covers the worst case where we have $\Theta = i_{n-1} < 0, i_{n-2} < i_{n-1}, \dots, i_0 < i_1$ by assigning least ordinal to i_0, the successor of that to i_1, and so on. More specifically, whenever we see a $i < M$ in Θ, we can assign i a value based on its position in Θ.

The second part also follows by induction on the derivation of $\left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right)\mathbf{seq}$, and relies on the fact that $\mathbb{W}_{\mathcal{S}}^{\mathbb{W}_{\mathcal{S}}} \sqsupseteq \mathbb{A}$ for all $\mathbb{T}_{\mathcal{S}}$-types $\mathbb{A}$ (Lemma 7.1), so the fact that both the positive and negative sides of $\mathbb{W}_{\mathcal{S}}^{\mathbb{W}_{\mathcal{S}}}$ are non-empty lets us choose something for every $x : A \in \Gamma$ and $\alpha : A \in \Delta$ regardless of A. □

Applications

Due to the parametric definition of the model in Section 7.3, the adequacy lemmas from Section 7.4 can be applied to a number of different goals. In particular, many standard properties in logic and programming languages—like logical consistency, type safety, strong normalization, and the soundness of extensionality—can be phrased as a particular application of adequacy by choosing the right notion of safety conditions and worlds.

In the following sections, we demonstrate each of these properties in turn. The general procedure for each proof is a three step process guided by the definition of the model:

1. Choose appropriate definitions for $\perp\!\!\!\perp$, $\mathbb{I}$, and $\mathbb{W}_{\mathcal{S}}$ and show that they result in a valid safety condition and worlds.

2. Find a valid substitution for the meaning of the well-formed sequent via Lemma 7.13, which may impose some restrictions on Γ and Δ.

3. Instantiate the semantic judgement via Lemma 7.12 with the substitution from step 2, proving that commands have the property asserted in $\perp\!\!\!\perp$.

Note that step 2, where we must exhibit there is a substitution for the well-formed sequent, is required to "unlock" the hypothetical semantic judgement from step 3. Thus, the substitutions can act as a gatekeeper, limiting which free (co-)variables in the input and output environments we can observe for that particular property.

Logical consistency

Logical consistency means there is no derivation of the empty sequent; i.e. there is no well-typed, closed command. This is the easiest application of the model, since every parameter is set to be as small or as big as possible. In particular, defining $\perp\!\!\!\perp = \emptyset$ is a valid choice, since then the only criteria of safety conditions (closure under expansion) is vacuously true.

To prove that we can come up with an empty instance of the model, we first show that the generativity requirement imposed by the worlds are trivially true in such an edge case.

Lemma 7.14 (Trivial generativity)**.** *For any safety condition $\mathcal{S} = (\mathbb{T}, \mathbb{I}, \perp\!\!\!\perp)$, and $\mathbb{T}$-spaces $\mathbb{U} = (\mathbb{U}_+, \mathbb{U}_-)$ and $\mathbb{V} = (\mathbb{V}_+, \mathbb{V}_-)$, and $\mathbb{I}$-space $\mathbb{W}$ such that $\mathbb{V}, \mathbb{W} \sqsubseteq \mathbb{U}$, if either $\mathbb{U}_+ = \mathbb{V}_+$, $\mathbb{U}_- = \mathbb{V}_-$, or $\perp\!\!\!\perp = \emptyset$, then all $\mathbb{I}$-spaces $\mathbb{A} = \mathbb{A}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}$ are $\perp\!\!\!\perp$-spaces.*

Proof. Let $\mathbb{W} = (\mathbb{W}_+, \mathbb{W}_-)$, let $\mathbb{A} = (\mathbb{A}_+, \mathbb{A}_-) = \mathbb{A}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}$, and let $v, e \in \mathbb{A}$. Under the assumption that $\mathbb{U}_+ = \mathbb{V}_+$, it must be that $v \in \mathbb{A}|_{\mathbb{V}}$ so that $\langle v \| e \rangle \in \perp\!\!\!\perp$ because $\mathbb{A} = \mathbb{A}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}$. Dually, under the assumption that $\mathbb{U}_- = \mathbb{V}_-$, it must be that $e \in \mathbb{A}|_{\mathbb{V}}$ so that $\langle v \| e \rangle \in \perp\!\!\!\perp$. Finally, under the assumption that $\perp\!\!\!\perp = \emptyset$, then either $\mathbb{A}_+ = \emptyset$ or $\mathbb{A}_- = \emptyset$ because $\mathbb{A} = \mathbb{A}|_{\mathbb{V}}^{\perp\!\!\!\perp_{\mathbb{W}}}$, so there is no way to have both $v \in \mathbb{A}$ and $e \in \mathbb{A}$ making the requirement that $\langle v \| e \rangle \in \perp\!\!\!\perp$ trivial. □

From the fact that generativity is trivial, we can show logical consistency of the underlying logic (in terms of the non-derivability of the empty sequent from Chapter III): there are no well-typed closed commands in $\mu\tilde{\mu}$.

Theorem 7.1 (Logical consistency)**.** *$c : \left(\vdash_{\mathcal{G}}^{\Theta} \right)$ is not derivable if $\left(\vdash_{\mathcal{G}}^{\Theta} \right)$* **seq** *is.*

Proof. We instantiate the model with $\mathbb{I} = \mathbb{T}$, $\perp\!\!\!\perp = \emptyset$ and $\mathbb{W}_{\mathcal{S}} = \mathbb{U}_{\mathcal{S}}$. Note that $\mathbb{S} = (\mathbb{T}, \mathbb{T}, \emptyset)$ is a safety condition since closure under expansion is trivially true, and likewise each $\mathbb{T}_{\mathcal{S}} = (\mathbb{U}_{\mathcal{S}}, \mathbb{V}_{\mathcal{S}}, \mathbb{U}_{\mathcal{S}})$ is an $\mathbb{S}$-world since it is trivially generative because of Lemma 7.14 and trivially saturated and focalizing because everything in $\mathbb{U}_{\mathcal{S}}$ is in $\mathbb{W}_{\mathcal{S}}$ by definition. Note that since $\rightsquigarrow$ is vacuous due to emptiness of $\perp\!\!\!\perp$, the actual evaluation strategies for $\vec{\mathcal{S}}$ are irrelevant.

Now, suppose that we have a derivation of $c : \left(\vdash^{\Theta}_{\mathcal{G}} \right)$. By adequacy (Lemma 7.12), it must be that $c : \left(\vDash^{\Theta}_{\mathcal{G}} \right)$. Additionally, supping a derivation of $\left(\vdash^{\Theta}_{\mathcal{G}} \right) \mathbf{seq}$ implies there is a $\sigma \in (\vDash) \, \mathbf{seq}$ by Lemma 7.13. Putting the two together, we get $c \{\pi_1 \circ \sigma\} \perp\!\!\!\perp c \{\pi_2 \circ \sigma\}$. But this is a contradiction since $\perp\!\!\!\perp$ is empty! So there can't be derivations of both $c : \left(\vdash^{\Theta}_{\mathcal{G}} \right)$ and $\left(\vdash^{\Theta}_{\mathcal{G}} \right) \mathbf{seq}$. □

Type safety

Now consider type safety, the property that well-typed programs don't go wrong (i.e. get stuck). We proved type safety via the syntactic technique of progress and preservation previously in Chapters III and IV, but we can also use the model to show a similar result for the much bigger language in Chapter VI in a way that is general over the chosen evaluation strategy. Type safety relies on a notion of "final commands" that represent the appropriate end states of a program, which we can characterize more broadly. For an arbitrary collection of strategies $\vec{\mathcal{S}}$, we say that a subset of commands, denoted by $\mathit{FinalCommand}_{\vec{\mathcal{S}}}$, is

- *necessary* if $c \not\mapsto_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}}$ for all $c \in \mathit{FinalCommand}_{\vec{\mathcal{S}}}$,
- *sufficient* if for all non-values $v :: \mathcal{S}$ either $\langle v \| e \rangle \mapsto_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}} c$ for some c or $\langle v \| e \rangle \in \mathit{FinalCommand}_{\vec{\mathcal{S}}}$ and for all non-co-values $e :: \mathcal{S}$ $\langle v \| e \rangle \mapsto_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}} c$ for some c or $\langle v \| e \rangle \in \mathit{FinalCommand}_{\vec{\mathcal{S}}}$, and
- *constructive* if $\left\langle \mathsf{K}^{\vec{C}}(\vec{E}, \vec{V}) \middle\| \alpha \right\rangle \in \mathit{FinalCommand}_{\vec{\mathcal{S}}}$ and $\left\langle x \middle\| \mathsf{O}^{\vec{C}}[\vec{V}, \vec{E}] \right\rangle \in \mathit{FinalCommand}_{\vec{\mathcal{S}}}$.

Also, note that there is no such thing as a well-typed closed command (as we saw during logical consistency), so we *must* allow for the ability to run some open commands. Thus, we characterize which free (co-)variables are allowed for a particular global environment $\mathcal{G}$. We say that an input environment Γ is *observable with respect to $\mathcal{G}$* if for all $(x : A) \in \Gamma$, $A = \mathsf{G}(\vec{C})$ where G is declared as a co-data type in $\mathcal{G}$.

Dually, we say that an output environment Δ is *observable with respect to* $\mathcal{G}$ if for all $(\alpha : A) \in \Delta$, $A = \mathsf{F}(\vec{C})$ where F is declared as a data type in $\mathcal{G}$. This follows the intuition that is okay to end with some (co-)data structure as the result, but we don't want to be blocked on a case analysis of some free (co-)variable. As a result, we can safely run any well-typed program typed in an observable environment without fear of getting stuck.

Theorem 7.2 (Type safety)**.** *For any focalizing strategies $\vec{S}$ such that $\mapsto_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}}$ is confluent, any necessary, sufficient, and constructive FinalCommand$_{\mathcal{S}}$, environments Γ and Δ that are observable with respect to $\mathcal{G}$, and derivations of $c : \left(\Gamma \vdash_{\mathcal{G}} \vec{\Delta}\right)$ and $\left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$* **seq** *, it follows that $c \mapsto\!\!\!\!\rightarrow_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}} c' \not\mapsto_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}}$ implies $c' \in$ FinalCommand$_{\mathcal{S}}$.*

Proof. We instantiate the model with

$$\begin{aligned}
\mathbb{I} &= \top\!\!\!\top \\
\perp\!\!\!\perp &= \{(c_1, c_2) \mid c_i \mapsto\!\!\!\!\rightarrow c_i' \not\mapsto \implies c_i \in \mathit{FinalCommand}_{\mathcal{S}}\} \\
\mathbb{W}_{\mathcal{S}} &= \mathbb{U}_{\mathcal{S}}
\end{aligned}$$

$\mathbb{S} = (\top\!\!\!\top, \top\!\!\!\top, \perp\!\!\!\perp)$ is a safety condition since $\perp\!\!\!\perp$ is closed under expansion by definition. Note that all $\perp\!\!\!\perp$-spaces $\mathbb{A}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$ are closed under $\mapsto$ reduction in the following sense:

- If $(v_1, v_2) \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$ and $(\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle) \rightsquigarrow (\langle v_1' \| e_1 \rangle, \langle v_2' \| e_2 \rangle)$ for any e_1 and e_2 then $(v_1', v_2') \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$ since $\langle v_1' \| e_1 \rangle \perp\!\!\!\perp \langle v_2' \| e_2 \rangle$ for all $e_1 \mathbb{A} e_2$ by confluence and the fact that *FinalCommand*$_{\mathcal{S}}$ is necessary so final commands do not reduce further by the operational semantics.

- Dually, if $(e_1, e_2) \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$ and $(\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle) \rightsquigarrow (\langle v_1 \| e_1' \rangle, \langle v_2 \| e_2' \rangle)$ for any v_1 and v_2 then $(e_1', e_2') \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$.

$\mathbb{T}_{\mathcal{S}} = (\mathbb{U}_{\mathcal{S}}, \mathbb{V}_{\mathcal{S}}, \mathbb{U}_{\mathcal{S}})$ is an $\mathbb{S}$-world since saturation and focalization is trivial and generation follows from the sufficiency of *FinalCommand*$_{\mathcal{S}}$: for all $(v_1, v_2), (e_1, e_2) \in \mathbb{A} = \mathbb{A}|_{\mathbb{V}_{\mathcal{S}}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$, if $v_i, e_i \notin \mathbb{V}_{\mathcal{S}}$ then one of the following holds

- $(\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle) \not\rightsquigarrow$ implies that $\langle v_1 \| e_1 \rangle \in$ *FinalCommand*$_{\mathcal{S}}$ and $\langle v_1 \| e_1 \rangle \in$ *FinalCommand*$_{\mathcal{S}}$ because *FinalCommand*$_{\mathcal{S}}$ is sufficient,

- $(\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle) \rightsquigarrow (\langle v_1' \| e_1 \rangle, \langle v_2' \| e_2 \rangle)$ implies that $(v_1', v_2') \in \mathbb{A}$ by forward closure mentioned above,

- $(\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle) \rightsquigarrow (\langle v_1 \| e'_1 \rangle, \langle v_2 \| e'_2 \rangle)$ implies that $(e'_1, e'_2) \in \mathbb{A}$ by forward closure mentioned above, or

- $(\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle) \rightsquigarrow (c_1, c_2)$ by some other means, can only happen when v_i or e_i is a (co-)value by the definition of $\mapsto_{\mu_{\vec{\mathcal{S}}} \tilde{\mu}_{\vec{\mathcal{S}}} \beta_{\vec{\mathcal{S}}} \varsigma_{\vec{\mathcal{S}}}}$.

So it can't happen that $\langle v_i \| e_i \rangle \mapsto\!\!\!\!\twoheadrightarrow c' \not\mapsto$ where $c' \notin \mathit{FinalCommand}_{\mathcal{S}}$.

Now, by Lemma 7.12, we have $c : \left(\Gamma \vDash_{\mathcal{G}} \Delta\right)$ and by Lemma 7.13, there is a $\sigma \in \left(\vDash_{\mathcal{G}} \right) \mathbf{seq}$. Note that for every data type $\mathsf{F}(\overrightarrow{X{:}k}) : \mathcal{SG}$, we have that $(\alpha, \alpha) \in \left[\!\left[\mathsf{F}(\vec{C})\right]\!\right]_{\pi_2 \circ \sigma}$ whenever $\left[\!\left[\mathsf{F}(\vec{C})\right]\!\right]_{\pi_2 \circ \sigma} \in [\![\mathcal{S}]\!]_{\pi_2 \circ \sigma}$ because $\mathit{FinalCommand}_{\vec{\mathcal{S}}}$ is constructive, so (α, α) is orthogonal to every construction of F. Also, for every co-data type $\mathsf{G}(\overrightarrow{X{:}k}) : \mathcal{S} \in \mathcal{G}$, we have that $(x, x) \in \left[\!\left[\mathsf{G}(\vec{C})\right]\!\right]_{\pi_2 \circ \sigma}$ whenever $\left[\!\left[\mathsf{G}(\vec{C})\right]\!\right]_{\pi_2 \circ \sigma} \in [\![\mathcal{S}]\!]_{\pi_2 \circ \sigma}$ for the dual reason. So because Γ and Δ are observable with respect to $\mathcal{G}$, we can extend $\sigma \in \left(\vDash_{\mathcal{G}} \right) \mathbf{seq}$ to $\sigma' \in \left(\Gamma \vDash_{\mathcal{G}} \Delta\right) \mathbf{seq}$ by substituting (co-)variables for themselves. And so $c = c\{\pi_1 \circ \sigma\} \perp\!\!\!\perp c\{\pi_2 \circ \sigma'\} = c$, meaning that $c \mapsto\!\!\!\!\twoheadrightarrow c' \not\mapsto$ implies $c' \in \mathit{FinalCommand}_{\vec{\mathcal{S}}}$. □

Note that the operational reduction relation $\mapsto$ is deterministic for both the $\mathcal{V}$ and $\mathcal{N}$ strategies, and while the $\mathcal{LV}$ and $\mathcal{LN}$ operational reduction relations aren't strictly deterministic—for example, in $\mathcal{LV}$ we can have multiple possible standard reductions as in

$$\langle x \| \tilde{\mu} x'. c\{y/y'\} \rangle \mapsfrom_{\tilde{\mu}_{\mathcal{LV}}} \langle x \| \tilde{\mu} x'. \langle y \| \tilde{\mu} y'. c \rangle \rangle \mapsto_{\tilde{\mu}_{\mathcal{LV}}} \langle y \| \tilde{\mu} y'. c\{x/x'\} \rangle$$

because both are evaluation contexts—they enjoy the diamond property, which implies confluence. Furthermore, we have the following set of final commands for the combination of four strategies defined as the smallest set $c_{fin} \in \mathit{FinalCommand}_{\mathcal{V},\mathcal{N},\mathcal{LV},\mathcal{LN}}$ such that:

- $\left\langle \mathsf{K}^{\vec{C}}(\vec{E}, \vec{V}) \middle\| \alpha \right\rangle \in \mathit{FinalCommand}_{\mathcal{V},\mathcal{N},\mathcal{LV},\mathcal{LN}}$,

- $\left\langle x \middle\| \mathsf{O}^{\vec{C}}[\vec{V}, \vec{E}] \right\rangle \in \mathit{FinalCommand}_{\mathcal{V},\mathcal{N},\mathcal{LV},\mathcal{LN}}$,

- $\langle v :: \mathcal{LV} \| \tilde{\mu} x{::}\mathcal{LV}. c_{fin} \rangle \in \mathit{FinalCommand}_{\mathcal{V},\mathcal{N},\mathcal{LV},\mathcal{LN}}$ if $v \notin \mathit{Value}_{\mathcal{LV}}$ and $\tilde{\mu} x{::}\mathcal{LV}. c_{fin} \notin \mathit{CoValue}_{\mathcal{N}}$, and

- $\langle \mu \alpha{::}\mathcal{LN}. c_{fin} \| e :: \mathcal{LN} \rangle \in \mathit{FinalCommand}_{\mathcal{V},\mathcal{N},\mathcal{LV},\mathcal{LN}}$ if $e \notin \mathit{CoValue}_{\mathcal{LN}}$ and $v \notin \mathit{Value}_{\mathcal{LN}}$

which is necessary, sufficient, and constructive. Therefore, we have type safety for any combination of these four strategies. Unfortunately, however, the non-deterministic strategy $\mathcal{U}$ is not covered by this application on the model which assumes at least a confluent operational semantics, and so it would require other techniques like the syntactic approach of progress and preservation.

Strong normalization

We can also demonstrate the strong normalization property: a command c, term v, and co-term e is *strongly normalizing* if there are no infinite reduction sequences starting from c, v, or e. This property relies heavily on the fact that $\mathbb{I}$ can be smaller than $\top\!\!\!\top$ and $\mathbb{W}_{\mathcal{S}}$ can be smaller than $\mathbb{U}_{\mathcal{S}}$. It also relies on the following *forward closure* property that types inherit from the safe pole $\perp\!\!\!\perp$.

Lemma 7.15 (Forward closure)**.** *For all safety conditions $\mathbb{S} = (\top\!\!\!\top, \mathbb{I}, \perp\!\!\!\perp)$, if $c \in \perp\!\!\!\perp$ and $c \to c' \in \perp\!\!\!\perp$ implies $c' \in \perp\!\!\!\perp$, then $v, e \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{B}}}$ and $v \to v'$ and $e \to e'$ implies $v', e' \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{B}}}$*

Proof. Since $v \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{B}}}$, we know $\langle v \| e \rangle \in \perp\!\!\!\perp$, and since $\langle v \| e \rangle \to \langle v' \| e \rangle$ we have $\langle v' \| e \rangle \in \perp\!\!\!\perp$ as well. Therefore, $v \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{B}}}$. The dual argument holds for $e \in \mathbb{A}^{\perp\!\!\!\perp_{\mathbb{B}}}$ where $e \to e'$. □

Strong normalization is different from type safety in that substitutions are much easier to come by. In particular, strong normalization is an observable property for *all* well-typed, open expressions. This is because (co-)variables act as benign (co-)values which can be safely paired with any strongly normalizing parter.

Lemma 7.16. *v and e are strongly normalizing if and only if $\langle v \| \alpha \rangle$ and $\langle x \| e \rangle$ are.*

Proof. Note that a command, term, or (co-)term is strongly normalizing if and only if all its reducts are, so we can perform noetherian induction on the well-founded (partial) order imposed by the possible reduction sequences starting from any strongly normalizing expression. When $\langle v \| \alpha \rangle$ is strongly normalizing than v is as well because it is a sub-term of $\langle v \| \alpha \rangle$, so any reduction of v is also a reduction inside $\langle v \| \alpha \rangle$. So suppose that v is strongly normalizing, and we can see that $\langle v \| \alpha \rangle$ is as well by noetherian induction on the possible reduction sequences starting from v:

– If $\langle v \| \alpha \rangle \to \langle v' \| \alpha \rangle$ because $v \to v'$, then $\langle v' \| \alpha \rangle$ is strongly normalizing by the inductive hypothesis.

- If $\langle v \| \alpha \rangle \to c\,\{\alpha/\beta\}$ because $v = \mu\beta.c$ then c is strongly normalizing because it is a sub-command of v, and $c\,\{\alpha/\beta\}$ is strongly normalizing as well since any reduction on $c\,\{\alpha/\beta\}$ is a corresponding reduction on c.

The fact that e is strongly normalizing if and only if $\langle x \| e \rangle$ is follows dually to the above. □

We therefore derive strong normalization for any combination of sufficiently deterministic, stable (Definition 6.1) and focalizing (Definition 5.2) strategies from adequacy by restricting $⫫$ and $\mathbb{W}_{\mathcal{S}}$ to only strongly normalizing expressions.

Theorem 7.3 (Strong normalization)**.** *Given any stable and focalizing substitution strategies $\mathcal{S}$ such that $\succ_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}}$ is deterministic, and derivations of $\left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$* **seq** *and $\Theta \vdash_{\mathcal{G}} A : k$,*

a) if $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then the weak $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in c,

b) if $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : \mid \Delta$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then the weak $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in v, and

c) if $\Gamma \mid e : \vdash^{\Theta}_{\mathcal{G}} \Delta$ is derivable in $\mu\tilde{\mu}_{\vec{\mathcal{S}}}$ then the weak $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{S}}\varsigma^{\mathcal{S}}\nu$ reduction theory is strongly normalizing in e.

Proof. We use $\mathit{Command}_{SN}$, Term_{SN}, and CoTerm_{SN} to denote the sets of strongly normalizing commands, terms, and co-terms, respectively, and instantiate the model as follows:

$$\begin{aligned}
\mathbb{W}_{\mathcal{S}} &= (\{(v, v') \in \mathbb{U}_{\mathcal{S}} \mid v, v' \in \mathit{Term}_{SN}\}, \{(e, e') \in \mathbb{U}_{\mathcal{S}} \mid e, e' \in \mathit{CoTerm}_{SN}\}) \\
⫫ &= \mathit{Command}_{SN} \times \mathit{Command}_{SN} \\
\mathbb{I} &= \{(\langle v \| e \rangle, \langle v' \| e' \rangle) \mid v, v' \in \mathit{Term}_{SN}, e, e' \in \mathit{CoTerm}_{Sn}\}
\end{aligned}$$

Furthermore, we will form evaluation strategies out of the given substitution strategies $\vec{\mathcal{S}}$ by using the minimal definition of *EvalCxt*s that includes only $\square$, $\langle \square \| E \rangle$, and $\langle V \| \square \rangle$

$\mathbb{S} = (\mathbb{T}, \mathbb{I}, ⫫)$ is a safety condition because it validates closure under expansion, which can be proved by noetherian induction over the strongly normalizing term and co-term components of commands in $\mathbb{I}$. Suppose that $(\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle) \in \mathbb{I}$ and $(\langle v_1 \| e_1 \rangle, \langle v_2 \| e_2 \rangle) \rightsquigarrow (c'_1, c'_2) \in ⫫$. If $\langle v_i \| e_i \rangle \succ c''_i$ then $c''_i = c'_i \in ⫫$ because $\succ_{\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}}$ is deterministic. If $\langle v_i \| e_i \rangle \to \langle v'_i \| e_i \rangle$ because $v_i \to v'_i$, then $\langle v'_i \| e_i \rangle \in \perp\!\!\!\perp$ by the inductive hypothesis. If $\langle v_i \| e_i \rangle \to \langle v_i \| e'_i \rangle$ because $e_i \to e'_i$, then $\langle v_i \| e'_i \rangle \in \perp\!\!\!\perp$ by the inductive hypothesis. Therefore, $\langle v_i \| e_i \rangle \in \perp\!\!\!\perp$.

Note that by Lemma 7.16, $\mathbb{W}_{\mathcal{S}} = (\mathit{Variable}, \mathit{CoVariable})^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$. Thus, $\mathbb{T}_{\mathcal{S}} = (\mathbb{U}_{\mathcal{S}}, \mathbb{V}_{\mathcal{S}}, \mathbb{W}_{\mathcal{S}})$ is an $\mathbb{S}$-world because it is saturated, generative, and focalizing as follows:

- *saturation:* $(\mathit{Variable}, \mathit{CoVariable}) \sqsubseteq \mathbb{W}_{\mathcal{S}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}\Big|_{\mathbb{V}_{\mathcal{S}}}$ by restricted double orthogonal introduction (Property 7.5 (c)) since (co-)variables are values. Thus, the possible $\langle v \| \alpha \rangle \mapsto c \in \mathit{Command}_{SN}$ are

  - $\langle \mu\beta.c \| \alpha \rangle \mapsto c\{\alpha/\beta\} \in \mathit{Command}_{SN}$, so that $c \in \mathit{Command}_{SN}$ and thus $\mu\beta.c \in \mathit{Term}_{SN}$.
  - $\left\langle \mathsf{K}^{\vec{C}}(\vec{E}, e', \vec{e}, \vec{v}) \middle\| \alpha \right\rangle \mapsto \left\langle \mu\alpha. \left\langle \mu\beta. \left\langle \mathsf{K}^{\vec{C}}(\vec{E}, \beta, \vec{e}, \vec{v}) \middle\| \alpha \right\rangle \middle\| e' \right\rangle \middle\| \alpha \right\rangle \in \mathit{Command}_{SN}$, where every reduction in $\mathsf{K}^{\vec{C}}(\vec{E}, e', \vec{e}, \vec{v})$ can be traced to a sub-(co-)-term in $\mu\alpha. \left\langle \mu\beta. \left\langle \mathsf{K}^{\vec{C}}(\vec{E}, \beta, \vec{e}, \vec{v}) \middle\| \alpha \right\rangle \middle\| e' \right\rangle$, except for the top-level ς reduction which is already known to lead to a strongly normalizing term.
  - $\left\langle \mathsf{K}^{\vec{C}}(\vec{E}, \vec{V}, v', \vec{v}) \middle\| \alpha \right\rangle \mapsto \left\langle \mu\alpha. \left\langle v' \middle\| \tilde{\mu}y. \left\langle \mathsf{K}^{\vec{C}}(\vec{E}, \vec{V}, y, \vec{v}) \middle\| \alpha \right\rangle \right\rangle \middle\| \alpha \right\rangle \in \mathit{Command}_{SN}$, is similar to the previous case.

  Likewise, for $\langle x \| e \rangle \mapsto c \in \mathit{Command}_{SN}$ $e \in \mathit{CoTerm}_{SN}$ by duality.

- *generation:* suppose that we have a $\mathbb{I}$-space $\mathbb{A} = \mathbb{A}|_{\mathbb{V}_{\mathcal{S}}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$ and let $(v_1, v_2), (e_1, e_2) \in \mathbb{A}$. The fact that $\langle v_i \| e_i \rangle \in \mathit{Command}_{SN}$ follows by noetherian induction on the possible reductions from v_i and e_i:

  - If $\langle v_i \| e_i \rangle \mapsto c'_i$ then one of v_i or e_i must be a (co-)value, so $\langle v_i \| e_i \rangle \in \mathit{Command}_{SN}$ already.
  - If $\langle v_i \| e_i \rangle \to \langle v'_i \| e_i \rangle$ because $v_i \to v'_i$, then $(v'_i, v_1) \in \mathbb{A}$ or $(v_1, v'_i) \in \mathbb{A}$ by Lemma 7.15, so $\langle v'_i \| e_i \rangle \in \mathit{Command}_{SN}$ by the inductive hypothesis.
  - If $\langle v_i \| e_i \rangle \to \langle v_i \| e'_i \rangle$ because $e_i \to e'_i$, then $\langle v'_i \| e_i \rangle \in \mathit{Command}_{SN}$ similarly to the previous case.

- *focalization:* structures (data structures and co-data observations) made from strongly normalizing (co-)values are strongly normalizing themselves, since the only possible reductions are with in those strongly normalizing (co-)values, and weak reduction does not reduce inside case abstractions, so they are trivially strongly normalizing.

Now, by Lemma 7.12, we have $c : \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Delta\right)$ and by Lemma 7.13, there is a $\sigma \in \left(\Gamma \vDash^{\Theta}_{\mathcal{G}} \Gamma\right)\mathbf{seq}$, since $\mathbb{W}_{\mathcal{S}} = (\mathit{Variable}, \mathit{CoVariable})^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$ and thus $(\mathit{Variable}, \mathit{CoVariable}) \sqsubseteq \mathbb{W}_{\mathcal{S}}|_{\mathbb{V}_{\mathcal{S}}}^{\perp\!\!\!\perp_{\mathbb{W}_{\mathcal{S}}}}$ by restricted double orthogonal introduction (Property 7.5 (c)). Therefore, $c\{\pi_1 \circ \sigma\} \perp\!\!\!\perp c\{\pi_2 \circ \sigma\}$, which implies that c is strongly normalizing. The instantiation of well-typed (co-)terms is similar. □

We only consider weak reduction, which does not reduce inside case abstractions, for simplicity since it makes the focalization requirement for case abstractions—that requires that they be strongly normalizing for all valid substitutions—trivial. However, we could extend the result to bounded reduction defined in Chapter VI by noting that there is at least one valid substitution so long as every quantified type of kind $< N$ is inhabited whenever N is ∞ or $M + 1$, as was done previously in (Downen *et al.*, 2015). Also note that each of the four strategies $\mathcal{V}$, $\mathcal{N}$, $\mathcal{LV}$, and $\mathcal{LLV}$ are stable and focalizing and have deterministic $\mu\tilde{\mu}$ rewriting rules, so any combination of them leads to a strongly normalizing reduction theory. But as with type safety, the non-deterministic $\mathcal{U}$ strategy is unfortunately not covered by this application of the model, because we need to assume determinacy of the primitive rewriting rules for demonstrating closure under expansion. To extend the strong normalization result to non-deterministic strategies like $\mathcal{U}$, we would need to utilize a different kind of model that better handles non-determinism like Barbanera & Berardi's (1994) symmetric candidates method.

Soundness of extensionality

Finally, we tackle the more difficult job of showing that the typed extensionality η axioms are sound with respect to the untyped operational semantics. Note that adequacy already justifies η in any instance of the model. So the only task left is to show that an appropriate notion of *contextual equivalence* is an instance of the model. The fact that equivalence relates two things, instead of being a predicate over a single thing, is the reason why we made the model binary instead of unary. Because we will be referring to the two different sets of rewrite rules, in this section we will use the shorthand $R_{\mathcal{S}}$ for the untyped rewrite rules $\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\beta_{\vec{\mathcal{S}}}\varsigma_{\vec{\mathcal{S}}}\nu$, and $E^{\mathcal{G}}_{\mathcal{S}}$ for the typed rules $\mu_{\vec{\mathcal{S}}}\tilde{\mu}_{\vec{\mathcal{S}}}\eta_{\mu}\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}\nu$.

First, we define a contextual equivalence relation for commands and (co-)terms of the $\mu\tilde{\mu}$-calculus based on the idea that we can distinguish the constructors or observers that were used to build (co-)data structures.

Definition 7.13 (Contextual equivalence)**.** For any strategies $\vec{\mathcal{S}}$, we say that two commands $c_1 : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ and $c_2 : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ are *contextually equivalent*, written $c_1 \cong_{\vec{\mathcal{S}}} c_2 : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$, if and only if for all environments Γ' and Δ' that are observable with respect to $\mathcal{G}$ and contexts C such that $C[c_1] : \left(\Gamma' \vdash_{\mathcal{G}} \Delta'\right)$ and $C[c_2] : \left(\Gamma' \vdash_{\mathcal{G}} \Delta'\right)$, $C[c_1] \mapsto\!\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}} D_1[c_1'] \not\mapsto\!\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}}$ if and only if $C[c_2] \mapsto\!\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}} D_2[c_2'] \not\mapsto\!\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}}$ for some $D_1, D_2 \in$ *EvalCxt*$_{\vec{\mathcal{S}}}$ and $c_1' \sim_\omega c_2'$, where $\sim_\omega$ is a weak equality on structures defined as:

$$\left\langle x \middle\| \mathsf{O}^{\vec{B}}[\vec{V}, \vec{E}] \right\rangle \sim_\omega \left\langle x \middle\| \mathsf{O}^{\vec{B'}}[\vec{V'}, \vec{E'}] \right\rangle \qquad \left\langle \mathsf{K}^{\vec{B}}(\vec{E}, \vec{V}) \middle\| \alpha \right\rangle \sim_\omega \left\langle \mathsf{K}^{\vec{B'}}(\vec{E'}, \vec{V'}) \middle\| \alpha \right\rangle$$

Furthermore, contextual equivalence is extended to term and (co-)terms as follows:

$$\Gamma \vdash^{\Theta}_{\mathcal{G}} v \cong_{\vec{\mathcal{S}}} v' : A \mid \Delta \triangleq \forall \Gamma' \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta',\ \langle v \| e \rangle \cong \langle v' \| e \rangle : \left(\Gamma', \Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta', \Delta\right)$$
$$\Gamma' \mid e \cong_{\vec{\mathcal{S}}} e' : A \vdash^{\Theta}_{\mathcal{G}} \Delta' \triangleq \forall \Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta,\ \langle v \| e \rangle \cong \langle v \| e' \rangle : \left(\Gamma', \Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta', \Delta\right)$$

The requirement that the constructor or observer matches in the final command is important to prevent contextual equivalence from collapsing: if $\iota_1\,() \cong_{\vec{\mathcal{S}}} \iota_2\,()$, for example, then everything is considered equivalent because:

$$c_1 \cong_{\vec{\mathcal{S}}} \langle \iota_1\,() \| \tilde{\mu}[\iota_1\,().c_1 \mid \iota_2\,().c_2] \rangle \cong_{\vec{\mathcal{S}}} \langle \iota_2\,() \| \tilde{\mu}[\iota_1\,().c_1 \mid \iota_2\,().c_2] \rangle \cong_{\vec{\mathcal{S}}} c_2$$

Thus, the requirement that the final constructors match means $\iota_1\,() \ncong_{\rightarrow} \iota_2\,()$.

Contextual equivalence is the “gold standard” equivalence relation for an operational semantics, as it is well-known that it is the biggest relation that is compatible (i.e. applies in any context) and doesn’t confuse answers.

Theorem 7.4 (Coarsest structure-preserving congruence)**.** *Suppose that $\mathcal{R}$ is a compatible relation between typed commands, terms, and co-terms such that for all environments Γ and Δ observable with respect to $\mathcal{G}$, and related commands $c_1 \mathrel{\mathcal{R}} c_2 : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$ implies $c_1 \mapsto\!\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}} D_1[c_1'] \not\mapsto\!\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}}$ if and only if $c_2 \mapsto\!\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}} D_2[c_2'] \not\mapsto\!\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}}$ for some $D_1, D_2 \in$ EvalCxt$_{\vec{\mathcal{S}}}$ and $c_1' \sim_\omega c_2'$. Then $\mathcal{R}$ is included in $\cong_{\vec{\mathcal{S}}}$.*

Proof. For the $\mathcal{R}$ relation on commands, suppose that $c_1 \mathrel{\mathcal{R}} c_2 : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ for arbitrary environments. Now, let Γ' and Δ' be observable with respect to $\mathcal{G}$ and

C be any context such that $C[c_i] : \left(\Gamma' \vdash_{\mathcal{G}} \Delta'\right)$. By the compatibility of $\mathcal{R}$, we have $C[c_1] \mathrel{\mathcal{R}} C[c_2] : \left(\Gamma' \vdash_{\mathcal{G}} \Delta'\right)$, and so by assumption $c_1 \mapsto\!\!\!\twoheadrightarrow_{R_{\vec{\mathcal{S}}}} D_1[c_1'] \not\mapsto_{R_{\vec{\mathcal{S}}}}$. Therefore, $c_1 \cong_{\vec{\mathcal{S}}} c_2 : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$. Similarly, compatibility of the (co-)term $\mathcal{R}$ relations means that if $\Gamma \vdash^{\Theta}_{\mathcal{G}} v \mathrel{\mathcal{R}} v' : A \mid \Delta$ and $\Gamma' \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta'$ then $\langle v \| e \rangle \mathrel{\mathcal{R}} \langle v' \| e \rangle : \left(\Gamma', \Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta', \Delta\right)$ and thus $\langle v \| e \rangle \cong_{\vec{\mathcal{S}}} \langle v' \| e \rangle : \left(\Gamma', \Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta', \Delta\right)$, and dually for co-terms. □

We can now show that the typed $\beta\eta$ theory of (co-)data is sound with respect to contextual equivalence of the untyped operational semantics for the polarized strategy $\mathcal{P} = \mathcal{V}, \mathcal{N}$ combining both call-by-value and call-by-name evaluation. As with type safety, we are limited in the Γ and Δ environments we can observe: so long as the variables in Γ only stand for co-data abstractions and the co-variables in Δ only stand for data abstractions, we cannot see the use of η in the result of the program when we only observe the top-level constructor or observer.

Theorem 7.5 (Soundness of $\beta\eta$ w.r.t. $\beta\varsigma$ in $\mathcal{P}$)**.** *Given environments* Γ *and* Δ *that are observable with respect to* $\mathcal{G}$*, and a derivation of* $\left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$ **seq** *, it follows that*

a) $c =_{E^{\mathcal{G}}_{\mathcal{P}}} c' : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$ *implies* $c \cong_{\mathcal{P}} c' : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right)$,

b) $v =_{E^{\mathcal{G}}_{\mathcal{P}}} v' : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right) A$ *implies* $v \cong_{\mathcal{P}} v' : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right) A$*, and*

c) $e =_{E^{\mathcal{G}}_{\mathcal{P}}} e' : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right) A$ *implies* $e \cong_{\mathcal{P}} e' : \left(\Gamma \vdash_{\mathcal{G}} \Delta\right) A$.

Proof. We instantiate the model as follows:

$$\begin{aligned}
\mathbb{I} &= \Top \\
\Perp &= \{(c_1, c_2) \in \Top \mid \exists c_1' \sim_\omega c_1',\ c_1 \mapsto\!\!\!\twoheadrightarrow_{R_{\mathcal{P}}} c_1' \iff c_2 \mapsto\!\!\!\twoheadrightarrow_{R_{\mathcal{P}}} c_2'\} \\
\mathbb{W}_{\mathcal{N}} &= \mathbb{U}_{\mathcal{N}} \\
\mathbb{W}_{\mathcal{V}} &= \mathbb{U}_{\mathcal{V}}
\end{aligned}$$

$\mathbb{S} = (\Top, \mathbb{I}, \Perp)$ is a safety condition because it is closed under expansion by definition. Furthermore, both $\mathbb{T}_{\mathcal{N}} = (\mathbb{U}_{\mathcal{N}}, \mathbb{V}_{\mathcal{N}}, \mathbb{W}_{\mathcal{N}})$ and $\mathbb{T}_{\mathcal{V}} = (\mathbb{U}_{\mathcal{V}}, \mathbb{V}_{\mathcal{V}}, \mathbb{W}_{\mathcal{V}})$ are $\mathbb{S}$-worlds because they are trivially saturated and focalizing since $\mathbb{W}_{\mathcal{S}} = \mathbb{U}_{\mathcal{S}}$, and trivially generative by Lemma 7.14.

Now, by Lemma 7.12, we have $c \Leftrightarrow c' : \left(\Gamma \vDash_{\mathcal{G}} \Delta\right)$ and by Lemma 7.13, there is a $\sigma \in \left(\vDash_{\mathcal{G}} \right)$ **seq** . Similar to the proof of type safety (Theorem 7.2), we show that variables are related by every co-data type and co-variables are related by every date type, letting us extend $\sigma \in \left(\vDash_{\mathcal{G}} \right) \mathbf{seq}$ to cover $\left(\Gamma \vDash_{\mathcal{G}} \Delta\right) \mathbf{seq}$. In particular, note that

$$\left\langle \mathsf{K}^{\vec{C}}(\vec{E},\vec{V}) \middle\| \alpha \right\rangle \mapsto\!\!\!\!\twoheadrightarrow_{R_{\mathcal{P}}} \left\langle \mathsf{K}^{\vec{C}}(\vec{E},\vec{V}) \middle\| \alpha \right\rangle \sim_{\omega} \left\langle \mathsf{K}^{\vec{C'}}(\vec{E'},\vec{V'}) \middle\| \alpha \right\rangle \twoheadleftarrow\!\!\!\!\dashv_{R_{\mathcal{P}}} \left\langle \mathsf{K}^{\vec{C'}}(\vec{E'},\vec{V'}) \middle\| \alpha \right\rangle$$

by reflexivity so $\left\langle \mathsf{K}^{\vec{C}}(\vec{E},\vec{V}) \middle\| \alpha \right\rangle \perp\!\!\!\perp \left\langle \mathsf{K}^{\vec{C'}}(\vec{E'},\vec{V'}) \middle\| \alpha \right\rangle$. Likewise $\left\langle y \middle\| \mathsf{O}^{\vec{C}}[\vec{V},\vec{E}] \right\rangle \perp\!\!\!\perp \left\langle y \middle\| \mathsf{O}^{\vec{C'}}[\vec{V'},\vec{E'}] \right\rangle$. Therefore, each co-data type relates y with y and each data type relates β with β by restricted double orthogonal introduction (Property 7.5 (c)), which means that we can extend $\sigma \in \left(\vDash_{\mathcal{G}} \right) \mathbf{seq}$ to $\sigma' \in \left(\Gamma \vDash_{\mathcal{G}} \Gamma\right) \mathbf{seq}$ by substituting all (co-)variables in Γ and Δ with themselves. Thus, we get that there is some $c_1 \sim_{\omega} c'_1$ such that $c = c\left\{\pi_1 \circ \sigma\right\} \mapsto\!\!\!\!\twoheadrightarrow_{R_{\mathcal{P}}} c_1$ if and only if $c' = c'\left\{\pi_2 \circ \sigma\right\} \mapsto\!\!\!\!\twoheadrightarrow_{R_{\mathcal{P}}} c'_1$. Therefore since $E^{\mathcal{G}}_{\mathcal{P}}$ equality is also compatible, we have that it is included by Theorem 7.4. The result for (co-)terms follows similarly, using the fact that for all semantic types $\mathbb{A}$ and $(v, v'), (e, e') \in \mathbb{A}$, $\langle v \| e \rangle \perp\!\!\!\perp \langle v' \| e' \rangle$. □

As a consequence of the soundness of the typed $\beta\eta$ equational theory with respect to the $\mathcal{P}$ instance of contextual equivalence means that $\beta\eta$ is *coherent* (i.e. there are at least two non-equal expression) in any observable context. In particular, previously we noted that $\iota_1\,()$ and $\iota_2\,()$ must be distinct terms with respect to contextual equivalence, which is forced by the fact that $\langle \iota_2\,() \| \alpha \rangle \not\sim_{\omega} \langle \iota_1\,() \| \alpha \rangle$. Therefore, since the $\beta\eta$-based equational theory is included in contextual equivalence by the above theorem, it must be that $\Gamma \vdash^{\Theta}_{\mathcal{G}} \iota_2\,() \neq_{\mu_{\mathcal{P}}\tilde{\mu}_{\mathcal{P}}\eta_{\mu}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} \iota_1\,() : A \oplus B \mid \Delta$ for any observable Γ and Δ, as that would imply $\Gamma \vdash^{\Theta}_{\mathcal{G}} \iota_2\,() \cong_{\mathcal{P}} \iota_1\,() : \ \mid \Delta$.

CHAPTER VIII

The Polar Basis for Types

How many types do programming languages really need? Mainstream statically typed programming languages all have some mechanisms for programmers to declare their own custom types to make writing software easier, so in practice there seems to be a limitless supply of different types in languages. However, when we are not working *in* a language but *with* a language—for example, to study its theoretical properties or to develop practical implementations—the fewer constructs and types the language has the easier it is to work with. This is where functional programming languages can shine by using their connection with logic to simplifying the language. For example, logic tells us that we only need a binary conjunction connective since larger conjunctions can be encoded by nesting the binary connective, and nothing is gained or lost because either nesting is equivalently provable: $(A \wedge B) \wedge C$ has a proof if an only if $A \wedge (B \wedge C)$ does. Similarly, an implication with multiple assumptions can be encoded by nesting the simpler single-assumption implication connective in the right way: $(A \wedge B) \supset C$ has a proof if and only if $A \supset (B \supset C)$ does. This lets us encode complex propositions and connectives in terms of a smaller number of more basic connectives. These encodings correspond to common techniques to simplify models of functional programming languages down to just a handful of primitive types—conjunction corresponds to pairing, so nested binary pairs can represent n-ary tuples, and implication corresponds to functions, so single-input functions can represent n-ary functions by *currying*—because the rest can be encoded by converting to and fro. But we don't just care if there are programs of the right types, we also care about what those programs *do*.

Unfortunately, in real functional languages these seemingly obvious encodings are not accurate because the two types do not actually describe the "same" set of program behaviors. If we want to say that a type is unnecessary because it can be encoded away, we should expect a one-for-one correspondence between the programs of both types, but this is often not the case because of the *effects* in the language. For example, encoding triples as nested pairs does not cause issue in a strict language like SML, but in a lazy language like Haskell we can observe a difference because of divergent (i.e. infinitely looping) or erroneous values like 1/0 or *undefined*. We can convert between nested pairs of type $(a,(b,c))$ and triples of type (a,b,c) as follows:

$$\begin{aligned}
&\mathit{fromTriple} && :: (a,b,c) \to (a,(b,c)) \\
&\mathit{fromTriple}\ (x,y,z) && = (x,(y,z)) \\
&\mathit{toTriple} && :: (a,(b,c)) \to (a,b,c) \\
&\mathit{toTriple}\quad p && = (\mathit{fst}\ p, \mathit{fst}\ (\mathit{snd}\ p), \mathit{snd}\ (\mathit{snd}\ p))
\end{aligned}$$

However, the nested pair type $(\mathit{Int},(\mathit{Bool},\mathit{String}))$ in Haskell contains both $(1,\mathit{loop}())$ and $(1,(\mathit{loop}(),\mathit{loop}()))$, where *loop* is a recursive function that loops forever and never returns an answer:

$$\begin{aligned}
&\mathit{loop} && :: a \to b \\
&\mathit{loop}\ x && = \mathit{loop}\ x
\end{aligned}$$

These two nested tuples can be distinguished by pattern-matching: for example, $\textbf{case}\ x\ \textbf{of}\ (\_,(\_,\_)) \to 9$ yields 9 when $x = (1,(\mathit{loop}(),\mathit{loop}()))$ but loops forever when $x = (1,\mathit{loop}())$. Yet the two different pairs are collapsed in the triple type (Int,b,c) which can only express $(1,\mathit{loop}(),\mathit{loop}())$, so a round trip to and from triples doesn't give back what we put in.[1]

The issues with unfaithful encodings are not just isolated to lazy languages; strict languages like SML have similar problems with different types. For example, the common technique known as *currying* in functional languages, which converts between functions of type $(a,b) \to c$ and $a \to (b \to c)$, lets us represent binary functions with unary functions nested in the right way as follows:

$$\mathit{curry}\ f\ x\ y = f\ (x,y) \qquad\qquad \mathit{uncurry}\ f\ (x,y) = f\ x\ y$$

However, this encoding too is not accurate. The type $a \to (b \to c)$ contains both functions $\lambda x.\mathit{loop}()$ and $\lambda x.\lambda y.\mathit{loop}()$ (where the non-terminating *loop* function raises is defined similarly in SML), which are observably different because the partial application $f\ 1$ loops forever when $f = \lambda x.\mathit{loop}()$ and returns a function when

[1] There is also the stricter alternative definition $\mathit{toTriple}\ (x,(y,z)) = (x,y,z)$, but a round-trip with this instead collapses $(5,\mathit{loop}())$ and $\mathit{loop}()$.

$f = \lambda x.\lambda y.loop()$. Yet, *uncurry* collapses these distinct values into $\lambda(x, y).loop()$, so a round-trip of *uncurry*ing and *curry*ing does not give back the same function.[2] Both of these counter-examples to simple, well-accepted encodings still cause trouble with infinite loops in place of exceptions.

The impact of inaccuracy on round-trips means that encodings have limited use in practice. Clearly we would prefer to have n-ary tuples instead of encoding them by hand in our source programming languages. Likewise, we want n-ary tuples in the target representation of lazy languages since they are more efficient to implement by reducing indirection and excessive thunking. So to utilize the above encodings in the middle of the compilation process, we need to go back and forth, and inaccurate encodings means that properties of the source and target language are lost: for example, the fact that $\lambda(x, y).f\ (x, y) = f$ in ML is lost by currying since $\lambda x.\lambda y.f\ x\ y \neq f$. So does this mean all hope is lost when our functional languages have effects, or even just general recursion? Must we choose between living with unfaithful encodings or giving up entirely on the game of encoding complex types into simpler primitives altogether?

Thankfully, we do not have to choose! The root cause of the unfaithfulness comes from a mismatch in the opportunities for strictness or laziness of programs that is implicit in types. This inherent connection between types and evaluation strategy (i.e. strictness and laziness) which we have seen previously under the guise of *polarity* (Zeilberger, 2009; Munch-Maccagnoni, 2013) in Chapter IV, which was originally developed in the setting of proof search not evaluation, as well as the *call-by-push-value* strategy (Levy, 2003). And it turns out that this fine-grained approach where programs can intermingle *both* strict and lazy evaluation gives us the tools we need to build strong encodings of types in languages with effects that accurately represent user-defined types.

For our setting, we will continue to model programs in the classical sequent calculus developed in the previous chapters (specifically Chapter V). This calculus has a built in control effect which makes the above issues of strictness and laziness relevant for encodings (because the root of the issue is programs which fail to return the expected result, regardless of whether the failure is caused by aborting the current

[2] Haskell also exhibits problems with currying, but instead due to differences in strictness on pairs. Specifically, $\lambda(\_, \_).9$ and $\lambda\_.9$ are different functions of type $(a, b) \to \mathit{Int}$ because they differ on the input $\bot :: (a, b)$, but these two functions are collapsed by a round-trip through currying and uncurrying (one way or the other, depending on the strictness of *uncurry*).

control path or an infinite loop), and lets us express a set of basic connectives with pleasant symmetries and relationships to one another.

This chapter covers the following topics:

- A primitive basis of polarized connectives suitable for encoding all simple (non-polymorphic and non-recursive) user-defined (co-)data types in languages with effects (Section 8.1).
- A definition of isomorphism between polarized types and simple (co-)data declarations in the sequent calculus (Section 8.2).
- A syntactic theory of isomorphisms for (co-)data types and their declarations (Section 8.3).
- A demonstration that the commonly expected algebraic and logical laws are sound (with respect to type isomorphism) for the primitive basis of polarized connectives (Section 8.4).
- An encoding of all simple user-defined (co-)data types in terms of the primitive basis, such that the encoded type is indeed isomorphic to the user-defined one (Section 8.5).

We stick to simple types for simplicity, to avoid the syntactic overhead of quantified type annotations from the higher-order parametric $\mu\tilde{\mu}$ sequent calculus from Section 6.2, and because they have a canonical finite basis in terms of polarized connectives. However, this development does not rely on simplicity, and could be generalized to accomodate polymorphic (co-)data declarations like those for the $\forall$ and $\exists$ connectives. Accommodating recursive types from Chapter VI, though, would require a different analysis.

Polarizing User-Defined (Co-)Data Types

We will express the polarized encoding problem—translating user-defined types into a fixed set of pre-defined primitive types—within the common framework of data and co-data in the parametric $\mu\tilde{\mu}$-sequent calculus. Our source language for representing user-defined types is the parametric $\mu\tilde{\mu}$-calculus from Chapter 5.2. In contrast to the λ-calculus, this setting makes it easier to include both call-by-name and call-by-value evaluation within the same program which lets us simultaneously model both ML- and Haskell-like languages within one framework, and to express additional types (like $A \parr B$ and $\sim A$ in Section 8.1) with pleasant symmetric properties (like the algebraic and logical laws in Section 8.4) that are generally not found in λ-based languages.

For example, to model pairs from functional languages, we can declare the following pair type for a kind $\mathcal{S}$—instantiating $\mathcal{V}$ for $\mathcal{S}$ for ML-like pairs and $\mathcal{N}$ for $\mathcal{S}$ for Haskell-like pairs—as follows to get the associated left and right rules:

$$\begin{aligned}
&\mathbf{data}\,(X : \mathcal{S}) \times_{\mathcal{S}} (Y : \mathcal{S}) : \mathcal{S}\,\mathbf{where} \\
&\qquad \mathsf{Pair}_{\mathcal{S}} : (X : \mathcal{S}, Y : \mathcal{S} \vdash X \times_{\mathcal{S}} Y \mid)
\end{aligned}$$

$$\frac{\Gamma \vdash_{\mathcal{G}} v_1 : A \mid \Delta \quad \Gamma' \vdash_{\mathcal{G}} v_2 : B \mid \Delta'}{\Gamma, \Gamma' \vdash_{\mathcal{G}} \mathsf{Pair}_{\mathcal{S}}(v_1, v_2) : A \times_{\mathcal{S}} B \mid \Delta, \Delta'} \times_{\mathcal{S}} R_{\mathsf{Pair}_{\mathcal{S}}}$$

$$\frac{c : \left(\Gamma, x_1 : A, x_2 : B \vdash_{\mathcal{G}} \Delta\right)}{\Gamma \mid \tilde{\mu}[\mathsf{Pair}_{\mathcal{S}}(x_1, x_2).c] : A \times_{\mathcal{S}} B \vdash_{\mathcal{G}} \Delta} \times_{\mathcal{S}} L$$

For this data type, the structure $\mathsf{Pair}_{\mathcal{S}}(v_1, v_2)$ is just like a pair from the respective functional language, and the usual case-analysis expression can be written as: $\mathbf{case}\, v\, \mathbf{of}\ \mathsf{Pair}_{\mathcal{S}}(x_1, x_2) \Rightarrow v' \triangleq \mu\alpha. \langle v \| \tilde{\mu}[\mathsf{Pair}_{\mathcal{S}}(x_1, x_2). \langle v \| \alpha \rangle] \rangle$. As another example, function types do not need to be primitives in this language, since they can be declared as co-data types. In particular, we have both strict and lazy functions by again instantiating the right kind for $\mathcal{S}$—picking $\mathcal{V}$ for strict functions and $\mathcal{N}$ for lazy ones—to get the associated left and right rules for functions:

$$\begin{aligned}
&\mathbf{codata}\,(X : \mathcal{S}) \to_{\mathcal{S}} (Y : \mathcal{S}) : \mathcal{S}\,\mathbf{where} \\
&\qquad \mathsf{Call}_{\mathcal{S}} : (X : \mathcal{S} \mid X \to_{\mathcal{S}} Y \vdash Y : \mathcal{S})
\end{aligned}$$

$$\frac{c : \left(\Gamma, x : A \vdash_{\mathcal{G}} \beta : B, \Delta\right)}{\Gamma \vdash_{\mathcal{G}} \mu(\mathsf{Call}_{\mathcal{S}}[x, \beta].c) : A \to_{\mathcal{S}} B \mid \Delta} \to_{\mathcal{S}} R$$

$$\frac{\Gamma \vdash_{\mathcal{G}} v : A \mid \Delta \quad \Gamma' \mid e : B \vdash_{\mathcal{G}} \Delta'}{\Gamma, \Gamma' \mid \mathsf{Call}_{\mathcal{S}}[v, e] : A \to_{\mathcal{S}} B \vdash_{\mathcal{G}} \Delta, \Delta'} \to_{\mathcal{S}} L_{\mathsf{Call}_{\mathcal{S}}}$$

As discussed in Chapter V, the familiar λ-calculus notation for function abstraction and application is written as:

$$\lambda x.v \triangleq \mu(\mathsf{Call}_{\mathcal{S}}[x,\beta].\langle v \| \beta \rangle) \qquad\qquad v\ v' \triangleq \mu\beta.\,\langle v \| \mathsf{Call}_{\mathcal{S}}[v',\beta]\rangle$$

Our target language, which we will use for encoding all the constructs from the source, is not really a different language at all. Rather, it is a limited subset of the source language, consisting of only a fixed number of different constructs. The idea is to declare just a handful of (co-)data types up front, collectively named $\mathcal{P}$, and then forget the declaration mechanism entirely to prevent the language from being extended with any new types. The key, then, is to ensure that all the programs from the source language can be *faithfully* encoded into the limited constructs included in the target, without running into the same troubles of unfaithful encodings from the introduction. In other words, we want to be able to faithfully encode any collection of declarations $\mathcal{G}$ into the pre-determined basis $\mathcal{P}$.

The brunt of our pre-defined primitive data and co-data types is given in Figure 8.1. Each of these (co-)data types are chosen as a basis because of their symmetry—for each one, there is a dual mirror image on the other side—and because they all perform one, and only one, aspect of the functionality allowed by the declaration mechanism. The additive (co-)data types capture the use of multiple different constructors or observers for a type by giving a choice between two ($\oplus$ and $\&$) or a choice of no (0 and $\top$) alternatives. The multiplicative (co-)data types capture the use of multiple components within structures or observations, by giving a combination of two ($\otimes$ and $⅋$) or no (1 and $\bot$) parts. And finally the negation (co-)data types, which capture the ability for data structures to contain co-terms and co-data observations to contain terms.

We might think that we have some flexibility in choosing the evaluation strategies for the declarations in Figure 8.1. But as it turns out, since we want to use these (co-)data types as the backbone of faithful encodings, our hand is forced. Intuitively, each of these declarations follows a simple rule of thumb for choosing the kinds for types: every type to the left (of $\vdash$) is $\mathcal{V}$ and every type to the right is $\mathcal{N}$, except for the active type whose kind is the reverse. This rule of thumb has a few consequences. The first is that every data type is call-by-value and every co-data type is call-by-name, which follows the general wisdom of *polarization* in computation (Zeilberger, 2009; Munch-Maccagnoni, 2013). The second consequence is that every data type constructor

Additive (co-)data types

$$\begin{array}{ll}
\mathbf{data}\,(X:\mathcal{V})\oplus(Y:\mathcal{V}):\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,(X:\mathcal{N})\mathbin{\&}(Y:\mathcal{N}):\mathcal{N}\,\mathbf{where} \\
\quad \iota_1:(X:\mathcal{V}\vdash X\oplus Y\mid\,) & \quad \pi_1:(\,\mid X\mathbin{\&}Y\vdash X:\mathcal{N}) \\
\quad \iota_2:(Y:\mathcal{V}\vdash X\oplus Y\mid\,) & \quad \pi_2:(\,\mid X\mathbin{\&}Y\vdash Y:\mathcal{N})
\end{array}$$

$$\begin{array}{ll}
\mathbf{data}\,0:\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,\top:\mathcal{N}\,\mathbf{where}
\end{array}$$

Multiplicative (co-)data types

$$\begin{array}{ll}
\mathbf{data}\,(X:\mathcal{V})\otimes(Y:\mathcal{V}):\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,(X:\mathcal{N})\parr(Y:\mathcal{N}):\mathcal{N}\,\mathbf{where} \\
\quad (\_,\_):(X:\mathcal{V},Y:\mathcal{V}\vdash X\otimes Y\mid\,) & \quad [\_,\_]:(\,\mid X\parr Y\vdash X:\mathcal{N},Y:\mathcal{N})
\end{array}$$

$$\begin{array}{ll}
\mathbf{data}\,1:\mathcal{V}\,\mathbf{where}\;\;():(\,\vdash 1\mid\,) & \mathbf{codata}\,\bot:\mathcal{N}\,\mathbf{where}\;\;[]:(\,\mid\bot\vdash\,)
\end{array}$$

Involutive negation (co-)data types

$$\begin{array}{ll}
\mathbf{data}\,{\sim}(X:\mathcal{N}):\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,\neg(X:\mathcal{V}):\mathcal{N}\,\mathbf{where} \\
\quad {\sim}:(\,\vdash{\sim}X\mid X:\mathcal{N}) & \quad \neg:(X:\mathcal{V}\mid\neg X\vdash\,)
\end{array}$$

FIGURE 8.1. Declarations of the primitive polarized data and co-data types.

builds on $\mathcal{V}$ types and every co-data type constructor builds on $\mathcal{N}$ types, except for the negation constructors which are reversed because their underlying (co-)terms are reversed. The last consequence is that the notion of data type values and co-data type co-values are hereditarily as restrictive as possible, where a structure or observation is only a (co-)value if it contains components that are (co-)values in the most restrictive sense.

The basic (co-)data types from Figure 8.1 are still incomplete, though, for our purpose of encoding *all* (co-)data types expressible in the source language. In particular, how could we possibly represent a type like the call-by-name pair $A \times_{\mathcal{N}} B$? The $\otimes$ data type constructor won't do since it operates over the wrong kind of types. Therefore, we need a mechanism for plainly "shifting" between $\mathcal{N}$ and $\mathcal{V}$ kinds of types, and to do that we must break our rule of thumb. One way to do the conversion is with singleton (co-)data types, declared as follows, that *wraps* a component of the

$$
\begin{array}{ll}
\mathbf{data}\,{\downarrow_{\mathcal{S}}}(X : \mathcal{S}) : \mathcal{V}\,\mathbf{where} & \mathbf{codata}\,{\uparrow_{\mathcal{S}}}(X : \mathcal{S}) : \mathcal{N}\,\mathbf{where} \\
\quad {\downarrow_{\mathcal{S}}} : (X : \mathcal{S} \vdash {\downarrow_{\mathcal{S}}} X \mid\,) & \quad {\uparrow_{\mathcal{S}}} : (\,\mid {\uparrow_{\mathcal{S}}} X \vdash X : \mathcal{S}) \\[1em]
\mathbf{data}\,{}_{\mathcal{S}}{\Uparrow}(X : \mathcal{V}) : \mathcal{S}\,\mathbf{where} & \mathbf{codata}\,{}_{\mathcal{S}}{\Downarrow}(X : \mathcal{N}) : \mathcal{S}\,\mathbf{where} \\
\quad {}_{\mathcal{S}}{\Uparrow} : (X : \mathcal{V} \vdash {}_{\mathcal{S}}{\Uparrow} X \mid\,) & \quad {}_{\mathcal{S}}{\Downarrow} : (\,\mid {}_{\mathcal{S}}{\Downarrow} X \vdash X : \mathcal{N})
\end{array}
$$

FIGURE 8.2. Declarations of the shifts between strategies as data and co-data types.

other strategy:

$$
\begin{array}{cc}
\mathbf{data}\,{\downarrow}(X : \mathcal{N}) : \mathcal{V}\,\mathbf{where} & \mathbf{codata}\,{\uparrow}(X : \mathcal{V}) : \mathcal{N}\,\mathbf{where} \\
{\downarrow} : (X : \mathcal{N} \vdash {\downarrow} X \mid\,) & {\uparrow} : (\,\mid {\uparrow} X \vdash X : \mathcal{V})
\end{array}
$$

The other possibility is a singleton (co-)data type that *is* of the other strategy, declared as follows:

$$
\begin{array}{cc}
\mathbf{codata}\,{\Downarrow}(X : \mathcal{N}) : \mathcal{V}\,\mathbf{where} & \mathbf{data}\,{\Uparrow}(X : \mathcal{V}) : \mathcal{N}\,\mathbf{where} \\
{\Downarrow} : (\,\mid {\Downarrow} X \vdash X : \mathcal{N}) & {\Uparrow} : (X : \mathcal{V} \vdash {\Uparrow} X \mid\,)
\end{array}
$$

As it turns out, we will use both styles of shifts because they are each useful in different situations for encoding complex (co-)data types. As a technical device, we will use a whole family of shifts parameterized by a kind of strategy as defined in Figure 8.2, with the above as defaults. The idea is that $\downarrow_{\mathcal{S}}$ and $\uparrow_{\mathcal{S}}$ shift *to* $\mathcal{V}$ and $\mathcal{N}$ (respectively) from $\mathcal{S}$, whereas ${}_{\mathcal{S}}{\Uparrow}$ and ${}_{\mathcal{S}}{\Downarrow}$ shift *from* $\mathcal{V}$ and $\mathcal{N}$ (respectively) to $\mathcal{S}$. These parameterized shifts include some redundancy (as we will see in Section 8.4), but they are useful notationally for generically manipulating types.

By combining the polarized types from Figure 8.1 with the shifts from Figure 8.2, we get the polarized basis $\mathcal{P}$ for all user-defined (co-)data types. In particular, the polarized basis is expressive enough to translate programs using any collection $\mathcal{G}$ of user-defined (co-)data types as shown in Figure 8.3, so that if $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$ then $[\![c]\!]_{\mathcal{G}} : ([\![\Gamma]\!]_{\mathcal{G}} \vdash^{\Theta}_{\mathcal{P}} [\![\Delta]\!]_{\mathcal{G}})$ (where $[\![\Gamma]\!]_{\mathcal{G}}$ and $[\![\Delta]\!]_{\mathcal{G}}$ are defined pointwise). We informally use deep pattern matching to aid writing the translation, with the understanding that it is desugared into several shallow patterns in the obvious way, and to express the repeated composition of the binary connectives, we define the ("big") versions of the polarized additive, multiplicative, and quantifier connectives over n-ary vectors of types and type variables as follows:

$$\bigoplus\epsilon \triangleq 0 \qquad \bigoplus(A,\vec{B}) \triangleq A \oplus \left(\bigoplus\vec{B}\right) \qquad \bigotimes\epsilon \triangleq 1 \qquad \bigotimes(A,\vec{B}) \triangleq A \otimes \left(\bigotimes\vec{B}\right)$$
$$\mathop{\&}\epsilon \triangleq \top \qquad \mathop{\&}(A,\vec{B}) \triangleq A \mathbin{\&} \left(\mathop{\&}\vec{B}\right) \qquad \bigparr\epsilon \triangleq \bot \qquad \bigparr(A,\vec{B}) \triangleq A \parr \left(\bigparr\vec{B}\right)$$

$$\iota_i\,(v) \triangleq \iota_2\,(.\overset{i}{.}.\iota_1\,(v)) \qquad\qquad \pi_i\,[e] \triangleq \pi_2\,[.\overset{i}{.}.\pi_1\,[e]]$$
$$(v_n,\dots,v_1) \triangleq (v_n,(\dots,(v_1,()))) \qquad\qquad [e_1,\dots,e_n] \triangleq [e_1,[\dots,[e_n,[]]]]$$

This polarizing encoding is *sound* so that equalities in the source, including η, are preserved in the target.

Theorem 8.1 (Polarization soundness)**.** *For any (composite) strategy $\mathcal{S}$ including $\mathcal{V}$ and $\mathcal{N}$, and $i = 1, 2$:*

a) if $c_i : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta)$ and $c_1 =_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} c_2$ then $[\![c_i]\!]_{\mathcal{G}} : ([\![\Gamma]\!]_{\mathcal{G}} \vdash^{\Theta}_{\mathcal{P}} [\![\Delta]\!]_{\mathcal{G}})$ and $[\![c_1]\!]_{\mathcal{G}} =_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{P}}\eta^{\mathcal{P}}} [\![c_2]\!]_{\mathcal{G}}$,

b) if $\Gamma \vdash^{\Theta}_{\mathcal{G}} v_i : A|\Delta$ $v_1 =_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} v_2$ then $[\![\Gamma]\!]_{\mathcal{G}} \vdash^{\Theta}_{\mathcal{P}} [\![v_i]\!]_{\mathcal{G}} : [\![A]\!]_{\mathcal{G}}|[\![\Delta]\!]_{\mathcal{G}}$ $[\![v_1]\!]_{\mathcal{G}} =_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{P}}\eta^{\mathcal{P}}} [\![v_2]\!]_{\mathcal{G}}$, and

c) if $\Gamma|e_i : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ and $e_1 =_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{G}}\eta^{\mathcal{G}}} e_2$ then $[\![\Gamma]\!]_{\mathcal{G}}|[\![e_i]\!]_{\mathcal{G}} : [\![A]\!]_{\mathcal{G}} \vdash^{\Theta}_{\mathcal{P}} [\![\Delta]\!]_{\mathcal{G}}$ and $[\![e_1]\!]_{\mathcal{G}} =_{\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_\mu\eta_{\tilde{\mu}}\beta^{\mathcal{P}}\eta^{\mathcal{P}}} [\![e_2]\!]_{\mathcal{G}}$.

Proof. The polarizing encoding of (co-)data types as shown in Figure 8.3 is stated in terms of deep pattern matching on data structures and co-data observations, which avoids the terrifying bureaucracy of the many levels of shallow patterns needed to implement the translation. Thankfully, these deep patterns fit a certain form which makes them much easier to desugar compared to fully general patterns. In particular, every pattern used in the encoding begins with a match on a ${}_{\mathcal{S}}\Uparrow$ or ${}_{\mathcal{S}}\Downarrow$ shift, then several nested matches on the additive structure of type $A \oplus B$ or $A \mathbin{\&} B$, and then

$$\llbracket X \rrbracket_{\mathcal{G}} \triangleq X$$

$$\llbracket \langle v \| e \rangle \rrbracket_{\mathcal{G}} \triangleq \langle \llbracket v \rrbracket_{\mathcal{G}} \| \llbracket e \rrbracket_{\mathcal{G}} \rangle$$

$$\llbracket x \rrbracket_{\mathcal{G}} \triangleq x \qquad\qquad \llbracket \alpha \rrbracket_{\mathcal{G}} \triangleq \alpha$$

$$\llbracket \mu\alpha.c \rrbracket_{\mathcal{G}} \triangleq \mu\alpha.\llbracket c \rrbracket_{\mathcal{G}} \quad \llbracket \tilde{\mu}x.c \rrbracket_{\mathcal{G}} \triangleq \tilde{\mu}x.\llbracket c \rrbracket_{\mathcal{G}}$$

Given $\mathbf{data}\,\mathsf{F}(\Theta) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{i1} : \mathcal{T}_{ij}}^{j} \vdash \mathsf{F}(\Theta) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$:

$$\left\llbracket \mathsf{F}(\vec{C}) \right\rrbracket_{\mathcal{G}} \triangleq {}_{\mathcal{S}}\Uparrow \left(\oplus \left(\overrightarrow{\otimes \left(\overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}} \llbracket B_{ij} \rrbracket_{\mathcal{G}} \theta)}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}} \llbracket A_{ij} \rrbracket_{\mathcal{G}} \theta}^{j} \right)}^{i} \right) \right)$$

$$\mathbf{where}\,\theta = \overrightarrow{\left\{ \llbracket C \rrbracket_{\mathcal{G}} / X \right\}}$$

$$\llbracket \mathsf{K}_i(\overrightarrow{e_{ij}}^{j}, \overrightarrow{v_{ij}}^{j}) \rrbracket_{\mathcal{G}} \triangleq {}_{\mathcal{S}}\Uparrow(\iota_i(\overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}}[\llbracket e_{ij} \rrbracket_{\mathcal{G}}])}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(\llbracket v_{ij} \rrbracket_{\mathcal{G}})}^{j}))$$

$$\left\llbracket \tilde{\mu}[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}).c_i}^{i}] \right\rrbracket_{\mathcal{G}} \triangleq \tilde{\mu}[\overrightarrow{{}_{\mathcal{S}}\Uparrow(\iota_i(\overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}])}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j})).\llbracket c_i \rrbracket_{\mathcal{G}}}^{i}]$$

Given $\mathbf{codata}\,\mathsf{G}(\Theta) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\Theta) \vdash \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$:

$$\left\llbracket \mathsf{G}(\vec{C}) \right\rrbracket_{\mathcal{G}} \triangleq {}_{\mathcal{S}}\Downarrow \left(\& \left(\overrightarrow{⅋ \left(\overrightarrow{\neg(\downarrow_{\mathcal{T}_{ij}} \llbracket A_{ij} \rrbracket_{\mathcal{G}} \theta)}^{j} \overrightarrow{\uparrow_{\mathcal{R}_{ij}} \llbracket B_{ij} \rrbracket_{\mathcal{G}} \theta}^{j} \right)}^{i} \right) \right)$$

$$\mathbf{where}\,\theta = \overrightarrow{\left\{ \llbracket C \rrbracket_{\mathcal{G}} / X \right\}}$$

$$\llbracket \mathsf{O}_i[\overrightarrow{v_{ij}}^{j}, \overrightarrow{e_{ij}}^{j}] \rrbracket_{\mathcal{G}} \triangleq {}_{\mathcal{S}}\Downarrow[\pi_i[\overrightarrow{\neg(\downarrow_{\mathcal{T}_{ij}}(\llbracket v_{ij} \rrbracket_{\mathcal{G}}))}^{j}, \overrightarrow{\uparrow_{\mathcal{R}_{i1}}[\llbracket e_{ij} \rrbracket_{\mathcal{G}}]}^{j}]]$$

$$\left\llbracket \mu(\overrightarrow{\mathsf{O}_i[\overrightarrow{x_{ij}}^{j}, \overrightarrow{\alpha_{ij}}^{j}].c_i}^{i}) \right\rrbracket_{\mathcal{G}} \triangleq \mu(\overrightarrow{{}_{\mathcal{S}}\Downarrow[\pi_i[\overrightarrow{\neg[\downarrow_{\mathcal{T}_{ij}}(x_{ij})]}^{j}, \overrightarrow{\uparrow_{\mathcal{T}_{ij}}[\alpha_{ij}]}^{j}]].\llbracket c_i \rrbracket_{\mathcal{G}}}^{i})$$

FIGURE 8.3. A polarizing translation from $\mathcal{G}$ into $\mathcal{P}$.

concludes with a match on the multiplicative structure of the following form:

$$\begin{aligned}
p \in \mathit{Pattern} &::= {}_{\mathcal{S}}{\Uparrow}\left(p^{+}\right) \\
p^{+} \in \mathit{AddPattern} &::= p^{\times} \mid \iota_1\left(p^{+}\right) \mid \iota_2\left(p^{+}\right) \\
p^{\times} \in \mathit{MultPattern} &::= x \mid () \mid \left(x, p^{\times}\right) \mid {\sim}\left(q^{\times}\right) \mid {\downarrow_{\mathcal{S}}}(x)
\end{aligned}$$

$$\begin{aligned}
q \in \mathit{CoPattern} &::= {}_{\mathcal{S}}{\Downarrow}\left[q^{+}\right] \\
q^{+} \in \mathit{AddCoPattern} &::= q^{\times} \mid \pi_1\left[q^{+}\right] \mid \pi_2\left[q^{+}\right] \\
q^{\times} \in \mathit{MultCoPattern} &::= \alpha \mid [] \mid \left[\alpha, q^{\times}\right] \mid \neg p^{\times} \mid {\uparrow_{\mathcal{S}}}[\alpha]
\end{aligned}$$

We can then easily desugar (co-)patterns of this form by just un-nesting the pattern one level at a time within the alternatives of every pattern matching (co-)term as follows:

$$\begin{aligned}
\tilde{\mu}\left[\overrightarrow{{}_{\mathcal{S}}{\Uparrow}\left(p_i^{+}\right).c_i}^{i}\right] &\triangleq \tilde{\mu}\left[{}_{\mathcal{S}}{\Uparrow}x.\left\langle x \middle\| \tilde{\mu}\left[\overrightarrow{p_i^{\exists}.c_i}^{i}\right]\right\rangle\right] \\
\tilde{\mu}\begin{bmatrix}\overrightarrow{\iota_1\left(p_i^{+}\right).c_i}^{i} \\ \overrightarrow{\iota_2\left(p_i'^{+}\right).c_i'}^{i}\end{bmatrix} &\triangleq \tilde{\mu}\begin{bmatrix}\iota_1\,(x).\left\langle x \middle\| \tilde{\mu}\left[\overrightarrow{p_i^{+}.c_i}^{i}\right]\right\rangle \\ \iota_2\,(x).\left\langle x \middle\| \tilde{\mu}\left[\overrightarrow{p_i'^{+}.c_i'}^{i}\right]\right\rangle\end{bmatrix} \\
\tilde{\mu}[x.c] &\triangleq \tilde{\mu}x.c \\
\tilde{\mu}\left[\left(y, p^{\times}\right).c\right] &\triangleq \tilde{\mu}\left[(y, x).\left\langle x \middle\| \tilde{\mu}\left[p^{\times}.c\right]\right\rangle\right] \\
\tilde{\mu}\left[{\sim}\left(q^{\times}\right).c\right] &\triangleq \tilde{\mu}\left[{\sim}(\alpha).\left\langle \mu\left(q^{\times}.c\right) \middle\| \alpha\right\rangle\right]
\end{aligned}$$

$$\begin{aligned}
\mu\left(\overrightarrow{{}_{\mathcal{S}}{\Downarrow}\left[q_i^{+}\right].c_i}^{i}\right) &\triangleq \mu\left({}_{\mathcal{S}}{\Downarrow}[\alpha].\left\langle \mu\left(\overrightarrow{q_i^{\forall}.c_i}^{i}\right) \middle\| \alpha\right\rangle\right) \\
\mu\begin{pmatrix}\overrightarrow{\pi_1\left[q_i^{+}\right].c_i}^{i} \\ \overrightarrow{\pi_2\left[q_i'^{+}\right].c_i'}^{i}\end{pmatrix} &\triangleq \mu\begin{pmatrix}\pi_1\,[\alpha].\left\langle \mu\left(\overrightarrow{q_i^{+}.c_i}^{i}\right) \middle\| \alpha\right\rangle \\ \pi_2\,[\alpha].\left\langle \mu\left(\overrightarrow{q_i'^{+}.c_i'}^{i}\right) \middle\| \alpha\right\rangle\end{pmatrix} \\
\mu(\alpha.c) &\triangleq \mu\alpha.c \\
\mu\left(\left[\beta, q^{\times}\right].c\right) &\triangleq \mu\left([\beta, \alpha].\left\langle \mu\left(q^{\times}.c\right) \middle\| \alpha\right\rangle\right) \\
\mu\left(\neg\left[p^{\times}\right].c\right) &\triangleq \mu\left(\neg\,[x].\left\langle x \middle\| \tilde{\mu}\left[p^{\times}.c\right]\right\rangle\right)
\end{aligned}$$

Additionally, in order to prove the soundness of the η law for (co-)data types with respect to the encoding, we use a couple helpful tricks with η. First, note that the seemingly stronger version of the η law for co-data types which applies to values (or the stronger η law for data types that applies to co-values)

$$
\begin{aligned}
&(\eta^{\mathsf{F}}_{\mathcal{S}}) && E : \mathsf{F}(\vec{C}) = \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).\langle \mathsf{K}(\vec{\alpha}, \vec{x}) \| E\rangle}\Big] \\
&(\eta^{\mathsf{G}}_{\mathcal{S}}) && V : \mathsf{G}(\vec{C}) = \mu\Big(\overrightarrow{\mathsf{O}[\vec{x}, \vec{\alpha}].\langle V \| \mathsf{O}[\vec{x}, \vec{\alpha}]\rangle}\Big)
\end{aligned}
$$

can be derived from the η law on (co-)variables by combining with the η_μ and $\eta_{\tilde{\mu}}$ rules for μ- and $\tilde{\mu}$-abstractions as follows:

$$
\begin{aligned}
E : \mathsf{F}(\vec{C}) &=_{\eta_\mu \eta_{\tilde{\mu}}} \tilde{\mu} y{:}\, \mathsf{F}(\vec{C}).\, \Big\langle \mu\beta{:}\, \mathsf{F}(\vec{C}).\, \langle y \| \beta\rangle \Big\| E \Big\rangle \\
&=_{\eta^{\mathsf{F}}} \tilde{\mu} y{:}\, \mathsf{F}(\vec{C}).\, \Big\langle \mu\beta{:}\, \mathsf{F}(\vec{C}).\, \Big\langle y \Big\| \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).\langle \mathsf{K}(\vec{\alpha}, \vec{x}) \| \beta\rangle}\Big] \Big\rangle \Big\| E \Big\rangle \\
&=_{\mu} \tilde{\mu} y{:}\, \mathsf{F}(\vec{C}).\, \Big\langle y \Big\| \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).\langle \mathsf{K}(\vec{\alpha}, \vec{x}) \| E\rangle}\Big] \Big\rangle \\
&=_{\eta_{\tilde{\mu}}} \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).\langle \mathsf{K}(\vec{\alpha}, \vec{x}) \| E\rangle}\Big]
\end{aligned}
$$

$$
\begin{aligned}
V : \mathsf{G}(\vec{C}) &=_{\eta_\mu \eta_{\tilde{\mu}}} \mu\beta{:}\, \mathsf{G}(\vec{C}).\, \Big\langle V \Big\| \tilde{\mu} y{:}\, \mathsf{G}(\vec{C}).\, \langle y \| \beta\rangle \Big\rangle \\
&=_{\eta^{\mathsf{G}}} \mu\beta{:}\, \mathsf{G}(\vec{C}).\, \Big\langle V \Big\| \tilde{\mu} y{:}\, \mathsf{G}(\vec{C}).\, \Big\langle \mu\Big(\overrightarrow{\mathsf{O}[\vec{x}, \vec{\alpha}].\langle y \| \mathsf{O}[\vec{x}, \vec{\alpha}]\rangle}\Big) \Big\| \beta \Big\rangle \Big\rangle \\
&=_{\tilde{\mu}} \mu\beta{:}\, \mathsf{G}(\vec{C}).\, \Big\langle \mu\Big(\overrightarrow{\mathsf{O}[\vec{x}, \vec{\alpha}].\langle V \| \mathsf{O}[\vec{x}, \vec{\alpha}]\rangle}\Big) \Big\| \beta \Big\rangle \\
&=_{\eta^{\mathsf{G}}} \mu\Big(\overrightarrow{\mathsf{O}[\vec{x}, \vec{\alpha}].\langle V \| \mathsf{O}[\vec{x}, \vec{\alpha}]\rangle}\Big)
\end{aligned}
$$

Second, note that we have the following equalities

$$
\begin{aligned}
&(\tilde{\mu}\eta^{\mathsf{F}}_{\mathcal{S}}) && c = \Big\langle z \Big\| \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c\,\{\mathsf{K}(\vec{\alpha}, \vec{x})/z\}}\Big] \Big\rangle && (\tilde{\mu} z.c : \mathsf{F}(\vec{C}) \in \mathit{CoValue}_{\mathcal{S}}) \\
&(\tilde{\mu}\eta^{\mathsf{G}}_{\mathcal{S}}) && c = \Big\langle \mu\Big(\overrightarrow{\mathsf{H}[\vec{x}, \vec{\alpha}].c\,\{\mathsf{H}[\vec{x}, \vec{\alpha}]/\gamma\}}\Big) \Big\| \gamma \Big\rangle && (\mu\gamma.c : \mathsf{G}(\vec{C}) \in \mathit{Value}_{\mathcal{S}})
\end{aligned}
$$

the first of which is derived from the η law of the F as follows:

$$
\begin{aligned}
c &=_{\tilde{\mu}_{\mathcal{S}}} \langle z \| \tilde{\mu} z.c\rangle \\
&=_{\eta^{\mathsf{F}}_{\mathcal{S}}} \Big\langle z \Big\| \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).\langle \mathsf{K}(\vec{\alpha}, \vec{x}) \| \tilde{\mu} z.c\rangle}\Big] \Big\rangle && (\tilde{\mu} z.c : \mathsf{F}(\vec{C}) \in \mathit{CoValue}_{\mathcal{S}}) \\
&=_{\tilde{\mu}_{\mathcal{S}}} \Big\langle z \Big\| \tilde{\mu}\Big[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c\,\{\mathsf{K}(\vec{\alpha}, \vec{x})/z\}}\Big] \Big\rangle
\end{aligned}
$$

and the second of which is likewise derived from the η law of G as follows:

$$
\begin{aligned}
c &=_{\mu_{\mathcal{S}}} \langle \mu\gamma.c \| \gamma\rangle \\
&=_{\eta^{\mathsf{G}}_{\mathcal{S}}} \Big\langle \mu\Big(\overrightarrow{\mathsf{H}[\vec{x}, \vec{\alpha}].\langle \mu\gamma.c \| \mathsf{H}[\vec{x}, \vec{\alpha}]\rangle}\Big) \Big\| \gamma \Big\rangle && (\mu\gamma.c : \mathsf{G}(\vec{C}) \in \mathit{Value}_{\mathcal{S}})
\end{aligned}
$$

$$=_{\mu_{\mathcal{S}}} \left\langle \mu\Big(\overrightarrow{\mathsf{H}[\vec{x},\vec{\alpha}].c\left\{\mathsf{H}[\vec{x},\vec{\alpha}]/\gamma\right\}}\Big) \middle\| \gamma \right\rangle$$

As examples, the particular instances of this rule for the polarized data types are:

$$\begin{array}{lll}
(\tilde{\mu}\eta^{\otimes}_{\mathcal{V}}) & c = \langle z \| \tilde{\mu}[(x,y).c\{(x,y)/z\}]\rangle & (z : A \otimes B) \\
(\tilde{\mu}\eta^{\oplus}_{\mathcal{V}}) & c = \langle z \| \tilde{\mu}[\iota_1(x).c\{\iota_1(x)/z\} | \iota_2(y).c\{\iota_2(y)/z\}]\rangle & (z : A \oplus B) \\
(\tilde{\mu}\eta^{1}_{\mathcal{V}}) & c = \langle z \| \tilde{\mu}[().c\{()/z\}]\rangle & (z : 1) \\
(\tilde{\mu}\eta^{0}_{\mathcal{V}}) & c = \langle z \| \tilde{\mu}[]\rangle & (z : 0) \\
(\tilde{\mu}\eta^{\sim}_{\mathcal{V}}) & c = \langle z \| \tilde{\mu}[\sim(\alpha).c\{\sim(\alpha)/z\}]\rangle & (z : {\sim}A)
\end{array}$$

and the instances for the polarized co-data types are:

$$\begin{array}{lll}
(\tilde{\mu}\eta^{⅋}_{\mathcal{N}}) & c = \langle \mu([\alpha,\beta].c\{[\alpha,\beta]/\gamma\}) \| \gamma\rangle & (\gamma : A ⅋ B) \\
(\tilde{\mu}\eta^{\&}_{\mathcal{N}}) & c = \langle \mu(\pi_1[\alpha].c\{\pi_1[\alpha]/\gamma\} | \pi_2[\beta].c\{\pi_2[\beta]/\gamma\}) \| \gamma\rangle & (\gamma : A \& B) \\
(\tilde{\mu}\eta^{\bot}_{\mathcal{N}}) & c = \langle \mu([].c\{[]/\gamma\}) \| \gamma\rangle & (\gamma : \bot) \\
(\tilde{\mu}\eta^{\top}_{\mathcal{N}}) & c = \langle \mu() \| \gamma\rangle & (\gamma : \top) \\
(\tilde{\mu}\eta^{\neg}_{\mathcal{N}}) & c = \langle \mu(\neg[x].c\{\neg[x]/\gamma\}) \| \gamma\rangle & (\gamma : \neg A)
\end{array}$$

With the above observations about pattern matching and extensionality, we are now ready to prove that the translation is sound. The fact that well-typed commands and (co-)terms have the associated translated type follows straightforwardly by (mutual) induction on their typing derivations. More interesting is the translation of equalities across the encoding. Note that since the translation is compositional and hygienic, the reflexive, symmetric, transitive, and (importantly) congruent closure of the equational theory is guaranteed (Downen & Ariola, 2014a). Therefore, we only need to check that each axiom is preserved by the translation. In that regard, it is important to note the fact that (1) that (co-)values translate to (co-)values and (2) substitution distributes over translation (that is, $[\![c]\!]_{\mathcal{G}}\left\{[\![V]\!]_{\mathcal{G}}/x\right\} =_{\alpha} [\![c\{V/x\}]\!]_{\mathcal{G}}$, *etc.*), both of which can be confirmed by induction on the syntax of (co-)terms.

The substitution axioms translate directly without change because of the above mentioned two facts about (co-)values and substitution, like so:

(η_μ) $\mu\alpha.\langle v\|\alpha\rangle = v$ translates to $[\![\mu\alpha.\langle v\|\alpha\rangle]\!]_{\mathcal{G}} \triangleq \mu\alpha.\langle[\![v]\!]_{\mathcal{G}}\|\alpha\rangle =_{\eta_\mu} [\![v]\!]_{\mathcal{G}}$

$(\eta_{\tilde{\mu}})$ $\tilde{\mu}x.\langle x\|e\rangle = e$ translates to $[\![\tilde{\mu}x.\langle x\|e\rangle]\!]_{\mathcal{G}} \triangleq \tilde{\mu}x.\langle x\|[\![e]\!]_{\mathcal{G}}\rangle =_{\eta_{\tilde{\mu}}} [\![e]\!]_{\mathcal{G}}$

(μ) $\langle \mu\alpha.c \| E \rangle = c\left\{E/\alpha\right\}$ translates to

$$\llbracket \langle \mu\alpha.c \| E \rangle \rrbracket \triangleq \left\langle \mu\alpha.\llbracket c \rrbracket_{\mathcal{G}} \middle\| \llbracket E \rrbracket_{\mathcal{G}} \right\rangle =_{\mu} \llbracket c \rrbracket_{\mathcal{G}} \left\{ \llbracket E \rrbracket_{\mathcal{G}}/\alpha \right\} =_{\alpha} \llbracket c\left\{E/\alpha\right\} \rrbracket_{\mathcal{G}}$$

$(\tilde{\mu})$ $\langle V \| \tilde{\mu}x.c \rangle = c\left\{V/x\right\}$ translates to

$$\llbracket \langle V \| \tilde{\mu}x.c \rangle \rrbracket \triangleq \left\langle \llbracket V \rrbracket_{\mathcal{G}} \middle\| \tilde{\mu}x.\llbracket c \rrbracket_{\mathcal{G}} \right\rangle =_{\tilde{\mu}} \llbracket c \rrbracket_{\mathcal{G}} \left\{ \llbracket V \rrbracket_{\mathcal{G}}/x \right\} =_{\alpha} \llbracket c\left\{V/x\right\} \rrbracket_{\mathcal{G}}$$

Given $\mathbf{data}\, \mathsf{F}(\Theta) : \mathcal{S}\, \mathbf{where}\, \overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{i1} : \mathcal{T}_{ij}}^{j} \vdash \mathsf{F}(\Theta) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j} \right)}^{i} \in \mathcal{G}$ we have:

(β^{F}) $\left\langle \mathsf{K}_i(\overrightarrow{e_{ij}}^{j}, \overrightarrow{v_{ij}}^{j}) \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y:k}}(\overrightarrow{\alpha_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}).c_i}^{i} \right] \right\rangle \;=\; \left\langle \mu\overrightarrow{\alpha_{ij}}^{j}.\left\langle \overrightarrow{v_{ij}} \middle\| \tilde{\mu}\overrightarrow{x_{ij}}^{j}.c_i \right\rangle \middle\| \overrightarrow{e_{ij}}^{j} \right\rangle$ translates by induction on the pattern ${}_{\mathcal{S}}\Uparrow\left(\iota_i \left(\overrightarrow{\neg\left[\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}] \right]}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j} \right) \right)$ to:

$$\begin{aligned}
&\left\llbracket \left\langle \mathsf{K}_i(\overrightarrow{e_{ij}}^{j}, \overrightarrow{v_{ij}}^{j}) \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}).c_i}^{i} \right] \right\rangle \right\rrbracket_{\mathcal{G}} \\
&\triangleq \left\langle {}_{\mathcal{S}}\Uparrow\left(\iota_i \left(\overrightarrow{\neg\left[\uparrow_{\mathcal{R}_{ij}}\left[\llbracket e_{ij} \rrbracket_{\mathcal{G}} \right] \right]}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}\left(\llbracket v_{ij} \rrbracket_{\mathcal{G}} \right)}^{j} \right) \right) \right. \\
&\qquad \left\| \tilde{\mu}\left[\overrightarrow{{}_{\mathcal{S}}\Uparrow\left(\iota_i \left(\overrightarrow{\neg\left[\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}] \right]}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j} \right) \right).\llbracket c_i \rrbracket_{\mathcal{G}}}^{i} \right] \right\rangle \\
&=_{\beta{\mathcal{S}\Uparrow}\eta_{\tilde{\mu}}} \left\langle \iota_i \left(\overrightarrow{\neg\left[\uparrow_{\mathcal{R}_{ij}}\left[\llbracket e_{ij} \rrbracket_{\mathcal{G}} \right] \right]}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}\left(\llbracket v_{ij} \rrbracket_{\mathcal{G}} \right)}^{j} \right) \middle\| \tilde{\mu}\left[\overrightarrow{\iota_i \left(\overrightarrow{\neg\left[\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}] \right]}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j} \right).\llbracket c_i \rrbracket_{\mathcal{G}}}^{i} \right] \right\rangle \\
&=_{\beta\oplus\eta_{\tilde{\mu}}} \left\langle \left(\overrightarrow{\neg\left[\uparrow_{\mathcal{R}_{ij}}\left[\llbracket e_{ij} \rrbracket_{\mathcal{G}} \right] \right]}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}\left(\llbracket v_{ij} \rrbracket_{\mathcal{G}} \right)}^{j} \right) \middle\| \tilde{\mu}\left[\left(\overrightarrow{\neg\left[\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}] \right]}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j} \right).\llbracket c_i \rrbracket_{\mathcal{G}} \right] \right\rangle \\
&\triangleq \left\langle \left(\neg\left[\uparrow_{\mathcal{R}_{i1}}\left[\llbracket e_{i1} \rrbracket_{\mathcal{G}} \right] \right], \dots, \neg\left[\uparrow_{\mathcal{R}_{im}}\left[\llbracket e_{im} \rrbracket_{\mathcal{G}} \right] \right], \overrightarrow{\downarrow_{\mathcal{T}_{ij}}\left(\llbracket v_{ij} \rrbracket_{\mathcal{G}} \right)}^{j} \right) \right. \\
&\qquad \left\| \tilde{\mu}\left[\left(\neg\left[\uparrow_{\mathcal{R}_{i1}}[\alpha_{i1}] \right], \dots, \neg\left[\uparrow_{\mathcal{R}_{im}}[\alpha_{im}] \right], \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j} \right).\llbracket c_i \rrbracket_{\mathcal{G}} \right] \right\rangle \\
&=_{\beta\otimes\eta_{\tilde{\mu}}} \left\langle \neg\left[\uparrow_{\mathcal{R}_{i1}}\left[\llbracket e_{i1} \rrbracket_{\mathcal{G}} \right] \right] \right| \\
&\qquad \left| \tilde{\mu}\left[\begin{array}{l} \neg\left[\uparrow_{\mathcal{R}_{i1}}[\alpha_{i1}] \right].\ \dots \\ \quad \left\langle \neg\left[\uparrow_{\mathcal{R}_{im}}\left[\llbracket e_{im} \rrbracket_{\mathcal{G}} \right] \right] \right| \\ \quad \left| \tilde{\mu}\left[\neg\left[\uparrow_{\mathcal{R}_{im}}[\alpha_{im}] \right].\left\langle \left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}}\left(\llbracket v_{ij} \rrbracket_{\mathcal{G}} \right)}^{j} \right) \middle\| \tilde{\mu}\left[\overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j}.\llbracket c_i \rrbracket_{\mathcal{G}} \right] \right\rangle \right] \right\rangle \end{array} \right] \right\rangle
\end{aligned}$$

$$
\begin{aligned}
&=_{\beta^{\neg}\eta_{\mu}} \left\langle \uparrow_{\mathcal{R}_{i1}}\left[\llbracket e_{i1}\rrbracket_{\mathcal{G}}\right] \middle| \right. \\
&\qquad \left. \middle| \tilde{\mu}\left[\begin{array}{l} \uparrow_{\mathcal{R}_{i1}}[\alpha_{i1}].\ \ldots \left\langle \uparrow_{\mathcal{R}_{im}}\left[\llbracket e_{im}\rrbracket_{\mathcal{G}}\right] \middle| \right. \\ \qquad\qquad\qquad \left. \middle| \tilde{\mu}\left[\uparrow_{\mathcal{R}_{im}}[\alpha_{im}].\left\langle \left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}}\left(\llbracket v_{ij}\rrbracket_{\mathcal{G}}\right)}^{j}\right) \middle\| \tilde{\mu}\left[\overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j}.\llbracket c_i\rrbracket_{\mathcal{G}}\right]\right\rangle\right]\right\rangle \end{array}\right]\right\rangle \\
&=_{\beta^{\uparrow}\eta_{\mu}} \left\langle \mu\alpha_{i1}.\ldots\left\langle \mu\alpha_{1m}.\left\langle \left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}}\left(\llbracket v_{ij}\rrbracket_{\mathcal{G}}\right)}^{j}\right) \middle\| \tilde{\mu}\left[\overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j}.\llbracket c_i\rrbracket_{\mathcal{G}}\right]\right\rangle \middle\| \llbracket e_{im}\rrbracket_{\mathcal{G}}\right\rangle \middle\| \llbracket e_{i1}\rrbracket_{\mathcal{G}}\right\rangle \\
&\triangleq \left\langle \mu\overrightarrow{\alpha_{ij}}^{j}.\left\langle \left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}}\left(\llbracket v_{ij}\rrbracket_{\mathcal{G}}\right)}^{j}\right) \middle\| \tilde{\mu}\left[\overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j}.\llbracket c_i\rrbracket_{\mathcal{G}}\right]\right\rangle \middle\| \overrightarrow{\llbracket e_{ij}\rrbracket_{\mathcal{G}}}^{j}\right\rangle \\
&=_{\beta^{\otimes}\beta^{1}\eta_{\tilde{\mu}}} \left\langle \mu\overrightarrow{\alpha_{ij}}^{j}.\left\langle \downarrow_{\mathcal{T}_{i1}}\left(\llbracket v_{i1}\rrbracket_{\mathcal{G}}\right) \middle\| \tilde{\mu}\left[\downarrow_{\mathcal{T}_{i1}}(x_{i1}).\ldots\left\langle \downarrow_{\mathcal{T}_{in}}\left(\llbracket v_{in}\rrbracket_{\mathcal{G}}\right) \middle\| \tilde{\mu}\left[\downarrow_{\mathcal{T}_{in}}(x_{in}).\llbracket c_i\rrbracket_{\mathcal{G}}\right]\right\rangle\right]\right\rangle \right. \\
&\qquad\qquad \left. \middle\| \overrightarrow{\llbracket e_{ij}\rrbracket_{\mathcal{G}}}^{j}\right\rangle \\
&=_{\beta^{\downarrow}\eta_{\tilde{\mu}}} \left\langle \mu\overrightarrow{\alpha_{ij}}^{j}.\left\langle \llbracket v_{i1}\rrbracket_{\mathcal{G}} \middle\| \tilde{\mu}x_{i1}.\ldots\left\langle \llbracket v_{in}\rrbracket_{\mathcal{G}} \middle\| \tilde{\mu}x_{in}.\llbracket c_i\rrbracket_{\mathcal{G}}\right\rangle\right\rangle \middle\| \overrightarrow{\llbracket e_{ij}\rrbracket_{\mathcal{G}}}^{j}\right\rangle \\
&\triangleq \left\langle \mu\overrightarrow{\alpha_{ij}}^{j}.\left\langle \overrightarrow{\llbracket v_{ij}\rrbracket_{\mathcal{G}}}^{j} \middle\| \tilde{\mu}\overrightarrow{x_{ij}}^{j}.\llbracket c_i\rrbracket_{\mathcal{G}}\right\rangle \middle\| \overrightarrow{\llbracket e_{ij}\rrbracket_{\mathcal{G}}}^{j}\right\rangle \\
&\triangleq \left\llbracket \left\langle \mu\overrightarrow{\alpha_{ij}}^{j}.\left\langle \overrightarrow{v_{ij}} \middle\| \tilde{\mu}\overrightarrow{x_{ij}}^{j}.c_i\right\rangle \middle\| \overrightarrow{e_{ij}}^{j}\right\rangle \right\rrbracket_{\mathcal{G}}
\end{aligned}
$$

(η^{F}) $\beta : \mathsf{F}(\overrightarrow{C}) = \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}).\left\langle \mathsf{K}_i(\overrightarrow{\alpha_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}) \middle\| \beta\right\rangle}^{i}\right]$ translates by induction on the pattern $\mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{\neg\left[\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}]\right]}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j}\right)\right)$ to:

$$
\begin{aligned}
&\left\llbracket \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}).\left\langle \mathsf{K}_i(\overrightarrow{\alpha_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}) \middle\| \beta\right\rangle}^{i}\right] \right\rrbracket_{\mathcal{G}} \\
&\triangleq \tilde{\mu}\left[\overrightarrow{\begin{array}{l} \mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{\sim\left(\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}]\right)}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j}\right)\right). \\ \qquad \left\langle \mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{\sim\left(\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}]\right)}^{j}, \overrightarrow{\downarrow_{\mathcal{T}_{ij}}(x_{ij})}^{j}\right)\right) \middle\| \beta\right\rangle \end{array}}^{i}\right] \\
&=_{\tilde{\mu}\eta_{\mathcal{V}}^{\downarrow}} \tilde{\mu}\left[\overrightarrow{\mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{\sim\left(\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}]\right)}^{j}, \overrightarrow{x_{ij}}^{j}\right)\right).\left\langle \mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{\sim\left(\uparrow_{\mathcal{R}_{ij}}[\alpha_{ij}]\right)}^{j}, \overrightarrow{x_{ij}}^{j}\right)\right) \middle\| \beta\right\rangle}^{i}\right] \\
&=_{\tilde{\mu}\eta_{\mathcal{V}}^{\uparrow}} \tilde{\mu}\left[\overrightarrow{\mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{\sim(\alpha_{ij})}^{j}, \overrightarrow{x_{ij}}^{j}\right)\right).\left\langle \mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{\sim(\alpha_{ij})}^{j}, \overrightarrow{x_{ij}}^{j}\right)\right) \middle\| \beta\right\rangle}^{i}\right] \\
&=_{\tilde{\mu}\eta_{\mathcal{V}}^{\sim}} \tilde{\mu}\left[\overrightarrow{\mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{y_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}\right)\right).\left\langle \mathcal{S}\Uparrow\left(\iota_i\left(\overrightarrow{y_{ij}}^{j}, \overrightarrow{x_{ij}}^{j}\right)\right) \middle\| \beta\right\rangle}^{i}\right] \\
&=_{\tilde{\mu}\eta_{\mathcal{V}}^{\otimes}\eta_{\mathcal{V}}^{1}} \tilde{\mu}\left[\overrightarrow{\mathcal{S}\Uparrow(\iota_i(x)).\langle \mathcal{S}\Uparrow(\iota_i(x)) \| \beta\rangle}^{i}\right]
\end{aligned}
$$

$$
\begin{aligned}
&=_{\tilde{\mu}\eta^{\oplus}_{\mathcal{V}}} \tilde{\mu}[{}_{\mathcal{S}}{\Uparrow}(x).\langle {}_{\mathcal{S}}{\Uparrow}(x) \| \beta \rangle] \\
&=_{\eta^{\Uparrow}} \beta
\end{aligned}
$$

Given **codata** $\mathsf{G}(\Theta) : \mathcal{S}$ **where** $\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\Theta) \vdash \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$ we have:

(β^{G}) is analogous to the translation of β^{F} by duality.

(η^{G}) is analogous to the translation of η^{F} by duality. □

But is the converse statement of completeness, that if the encodings of two commands or (co-)terms are equal then they are equal to begin with, also true? Unfortunately not so directly; the polarizing encoding has the effect of “anonymizing” types by moving away from a nominal style, where the different declarations lead to distinct types, to a more structural style, where differently declared types can be collapsed if they share a common underlying pattern of (co-)term structures. This collapse of types doesn’t mean that all hope is lost, however, because the (co-)terms are only collapsed *between* types not *within* types; there is still a one-for-one correspondence between typed (co-)terms of the same type in the source with the encoded (co-)terms in the target. To argue this case, we turn to applying the idea of *isomorphisms* between types (Di Cosmo, 1995).

Type Isomorphisms

Usually, we can say that two types are isomorphic when there are mappings to and from both of them whose composition is the identity. In the sequent setting, we interpret “mappings” as open commands with a free variable and co-variable, and the “identity” mapping is the simple command $\langle x \| \alpha \rangle$ connecting its free (co-)variables.

Definition 8.1 (Type isomorphism)**.** Two closed types A and B are *isomorphic*, written $A \approx B$, if and only if there exist commands $c : (x : A \vdash \beta : B)$ and $c' : (y : B \vdash \alpha : A)$ for any x, y, α, β such that the following equalities hold:

$$
\langle \mu\beta.c \| \tilde{\mu}y.c' \rangle = \langle x \| \alpha \rangle : (x : A \vdash \alpha : A) \qquad \langle \mu\alpha.c' \| \tilde{\mu}x.c \rangle = \langle y \| \beta \rangle : (y : B \vdash \beta : B)
$$

Moreover, two open types A and B with free type variables $\overrightarrow{X : \mathcal{S}}$ are *isomorphic*, written $\overrightarrow{X : \mathcal{S}} \vDash A \approx B$, if and only if for all types $\overrightarrow{C : \mathcal{S}}$, it follows that $A\{\overrightarrow{C/X}\} \approx B\{\overrightarrow{C/X}\}$.

Note that this definition of isomorphism between types is equivalent to a more traditional presentation in terms of inverse functions within the language. In particular, two types $A : \mathcal{S}$ and $B : \mathcal{S}$ are isomorphic in the sense of Definition 8.1 if and only if there are two closed function values $V : A \to_{\mathcal{S}} B$ and $V' : B \to_{\mathcal{S}} A$ such that $V' \circ V = \mathit{id} : A \to_{\mathcal{S}} A$ and $V \circ V' = \mathit{id} : B \to_{\mathcal{S}} B$, because we can always abstract over the open commands to get a pair of closed functions, or call the functions to retrieve a pair of open commands, where one is inverse whenever the other is. However, Definition 8.1 has the advantage of not assuming that our language has a primitive function type (since they are just user-defined co-data types like any other), and of avoiding the awkwardness of mapping between the different kinds of types that might be isomorphic to one another.

Having defined what is an isomorphism between types, we should ask if it is actually an equivalence relation as expected; are type isomorphisms closed under reflexivity, symmetry, and transitivity? The reflexivity and symmetry of the isomorphism relation between types is rather straightforward to show.

Theorem 8.2 (Reflexivity and Symmetry)**.** *For all types A and B, (a) $A \approx A$, and (b) $A \approx B$ implies $B \approx A$.*

Proof. The symmetry of type isomorphism follows immediately from its symmetric definition. More interestingly, we can establish the reflexive isomorphism of any type with the extensionality laws of μ- and $\tilde{\mu}$-abstractions. In particular, for a given A, we have the command $\langle x \| \alpha \rangle : (x : A \vdash \alpha : A)$ which serves as both open commands of the isomorphism $A \approx A$. The fact that the self-composition of this command is equal to itself comes from the η_μ and $\eta_{\tilde{\mu}}$ axioms: $\langle \mu\alpha.\, \langle x \| \alpha \rangle \| \tilde{\mu}x.\, \langle x \| \alpha \rangle \rangle =_{\eta_\mu} \langle x \| \tilde{\mu}x.\, \langle x \| \alpha \rangle \rangle =_{\eta_{\tilde{\mu}}} \langle x \| \alpha \rangle$. □

In contrast, transitivity of type isomorphisms is tricker, and in fact it is not guaranteed to hold in every possible situation. In particular, the transitivity of isomorphism relies on the exchange of μ- and $\tilde{\mu}$-bindings, which reassociates the composition of commands, but this not always valid in the multiple-strategy scenario. Specifically, given two any two (co-)terms of different kinds (recall the system for distinguishing between multiple base kinds in Figures 5.17 and 5.18 from Section 5.5), $v :: \mathcal{S}$ and $e :: \mathcal{T}$, the *exchange* law $\chi_{\mathcal{S} \vdash \mathcal{T}}$ is:

$$(\chi_{\mathcal{S} \vdash \mathcal{T}}) \qquad \langle v \| \tilde{\mu}x{::}\mathcal{S}.\, \langle \mu\alpha{::}\mathcal{T}.c \| e \rangle \rangle = \langle \mu\alpha{::}\mathcal{T}.\, \langle v \| \tilde{\mu}x{::}\mathcal{S}.c \rangle \| e \rangle$$

And when exchanging bindings of the same $\mathcal{S}$, we just write $\chi_{\mathcal{S}}$ for $\chi_{\mathcal{S}\vdash\mathcal{S}}$. Thankfully, even though exchange is not necessarily guaranteed, it is still valid for many combinations of strategies. For any $\mathcal{S}$, $\chi_{\mathcal{N}\vdash\mathcal{S}}$ is derivable from the universal strength of the $\tilde{\mu}_{\mathcal{N}}$ axiom and likewise $\chi_{\mathcal{S}\vdash\mathcal{V}}$ is derivable from the $\mu_{\mathcal{V}}$ axiom. So for all combinations of $\mathcal{N}$ and $\mathcal{V}$, each of $\chi_{\mathcal{N}\vdash\mathcal{N}}$, $\chi_{\mathcal{V}\vdash\mathcal{V}}$, and $\chi_{\mathcal{N}\vdash\mathcal{V}}$ hold, but $\chi_{\mathcal{V}\vdash\mathcal{N}}$ is invalidated by the counter example:[3]

$$\begin{aligned}
&\langle \mu\_{::}\mathcal{V}.c_1 \| \tilde{\mu}x{::}\mathcal{V}.\, \langle \mu\alpha{::}\mathcal{N}.c \| \tilde{\mu}\_{:}B{:}\mathcal{N}.c_2 \rangle\rangle =_{\mu_{\mathcal{V}}} c_1 \\
&\neq c_2 =_{\tilde{\mu}_{\mathcal{N}}} \langle \mu\alpha{::}\mathcal{N}.\, \langle \mu\_{::}\mathcal{V}.c_1 \| \tilde{\mu}x{::}\mathcal{V}.c\rangle \| \tilde{\mu}\_{::}\mathcal{N}.c_2\rangle
\end{aligned}$$

As it turns out, transitivity of type isomorphisms can be built on χ. The consequence of $\chi_{\mathcal{V}\vdash\mathcal{N}}$ being invalid is that, in general, isomorphisms between types of the same kind $\mathcal{S}$ are always transitive because $\chi_{\mathcal{S}}$ holds, but isomorphisms between different kinds of types might not be because we can't rely on *both* $\chi_{\mathcal{S}\vdash\mathcal{T}}$ and $\chi_{\mathcal{T}\vdash\mathcal{S}}$.

Theorem 8.3 (Homogeneous transitivity)**.** *For all strategies $\mathcal{S}$ such that $\chi_{\mathcal{S}}$ holds and types $A : \mathcal{S}$, $B : \mathcal{S}$, and $C : \mathcal{S}$, if $A \approx B$ and $B \approx C$ then $A \approx C$.*

Proof. Let $c_1 : (x : A \vdash \beta : B)$ and $c_2 : (y : B \vdash \alpha : A)$ be the commands from the isomorphism $A \approx B$, and let $c_3 : (y' : B \vdash \gamma : C)$ and $c_4 : (z : C \vdash \beta' : B)$ be the commands from the isomorphism $B \approx C$. We now establish the isomorphism $A \approx C$ by composing c_1 and c_3 to get c_5, and composing c_2 and c_4 to get c_6:

$$c_5 \triangleq \langle \mu\beta.c_1 \| \tilde{\mu}y'.c_3\rangle : (x : A \vdash \gamma : C) \qquad c_6 \triangleq \langle \mu\beta'.c_4 \| \tilde{\mu}y.c_2\rangle : (z : C \vdash \alpha : A)$$

With the help of $\chi_{\mathcal{S}}$, we get that the composition of c_5 and c_6 is the identity command $\langle x \| \alpha\rangle : (x : A \vdash \alpha : A)$:

$$\begin{aligned}
\langle \mu\gamma.c_5 \| \tilde{\mu}z.c_6\rangle &\triangleq \langle \mu\gamma.\, \langle \mu\beta.c_1 \| \tilde{\mu}y'.c_3\rangle \| \tilde{\mu}z.\, \langle \mu\beta'.c_4 \| \tilde{\mu}y.c_2\rangle\rangle \\
&=_{\chi_{\mathcal{S}}} \langle \mu\beta'.\, \langle \mu\gamma.\, \langle \mu\beta.c_1 \| \tilde{\mu}y'.c_3\rangle \| \tilde{\mu}z.c_4\rangle \| \tilde{\mu}y.c_2\rangle \\
&=_{\chi_{\mathcal{S}}} \langle \mu\beta'.\, \langle \mu\beta.c_1 \| \tilde{\mu}y'.\, \langle \mu\gamma.c_3 \| \tilde{\mu}z.c_4\rangle\rangle \| \tilde{\mu}y.c_2\rangle \\
&=_{Iso} \langle \mu\beta'.\, \langle \mu\beta.c_1 \| \tilde{\mu}y'.\, \langle y' \| \beta'\rangle\rangle \| \tilde{\mu}y.c_2\rangle \\
&=_{\eta_{\tilde{\mu}}} \langle \mu\beta'.\, \langle \mu\beta.c_1 \| \beta'\rangle \| \tilde{\mu}y.c_2\rangle
\end{aligned}$$

[3] The invalidity of $\chi_{\mathcal{V}\vdash\mathcal{N}}$ exactly corresponds to the failure of associativity in categorical models of polarity Munch-Maccagnoni (2013).

$$=_{\eta_\mu} \langle \mu\beta.c_1 \| \tilde{\mu}y.c_2 \rangle =_{Iso} \langle x \| \alpha \rangle$$

In the other direction, the composition of c_6 and c_5 is the identity command $\langle z \| \gamma \rangle$: $(z : C \vdash \gamma : C)$:

$$\begin{aligned}
\langle \mu\alpha.c_6 \| \tilde{\mu}x.c_5 \rangle &\triangleq \langle \mu\alpha. \langle \mu\beta'.c_4 \| \tilde{\mu}y.c_2 \rangle \| \tilde{\mu}x. \langle \mu\beta.c_1 \| \tilde{\mu}y'.c_3 \rangle \rangle \\
&=_{\chi_{\mathcal{S}}} \langle \mu\beta. \langle \mu\alpha. \langle \mu\beta'.c_4 \| \tilde{\mu}y.c_2 \rangle \| \tilde{\mu}x.c_1 \rangle \| \tilde{\mu}y'.c_3 \rangle \\
&=_{\chi_{\mathcal{S}}} \langle \mu\beta. \langle \mu\beta'.c_4 \| \tilde{\mu}y. \langle \mu\alpha.c_2 \| \tilde{\mu}x.c_1 \rangle \rangle \| \tilde{\mu}y'.c_3 \rangle \\
&=_{Iso} \langle \mu\beta. \langle \mu\beta'.c_4 \| \tilde{\mu}y. \langle y \| \beta \rangle \rangle \| \tilde{\mu}y'.c_3 \rangle \\
&=_{\eta_{\tilde{\mu}}} \langle \mu\beta. \langle \mu\beta'.c_4 \| \beta \rangle \| \tilde{\mu}y'.c_3 \rangle \\
&=_{\eta_\mu} \langle \mu\beta'.c_4 \| \tilde{\mu}y'.c_3 \rangle =_{Iso} \langle z \| \gamma \rangle
\end{aligned}$$

□

So isomorphisms between types of the same kind *are* an equivalence relation. But do they give us the right sense of a one-for-one correspondence between (co-)terms of those types? As it turns out, an isomorphism $A \approx B$ of $\mathcal{S}$-kinded types provides just enough structure to convert all equalities between A-typed (co-)terms to B-typed (co-)terms, and vice versa, which also relies on the $\chi_{\mathcal{S}}$ axiom to exchange (co-)variable bindings.

Theorem 8.4. *For all strategies $\mathcal{S}$ such that $\chi_{\mathcal{S}}$, types $A : \mathcal{S} \approx B : \mathcal{S}$, and environments Γ and Δ, there are contexts C and C' such that if $\Gamma \vdash_{\mathcal{G}} v_i : A \mid \Delta$ and $\Gamma \vdash_{\mathcal{G}} e_i : A \mid \Delta$ (for $i = 1, 2$), then $\Gamma \vdash_{\mathcal{G}} C[v_i] : B \mid \Delta$ and $\Gamma \vdash_{\mathcal{G}} C'[e_i] : B \mid \Delta$ (for $i = 1, 2$), $v_1 = v_2$ if and only if $C[v_1] = C[v_2]$, and $e_1 = e_2$ if and only if $C'[e_1] = C'[e_2]$.*

Proof. Let $c : (x : A \vdash \beta : B)$ and $c' : (y : B \vdash \alpha : A)$ be the commands from the isomorphism $A \approx B$, where $x, y \notin \Gamma$ and $\alpha, \beta \notin \Delta$ (renaming c and c' as necessary). The contexts are then $C \triangleq \mu\beta. \langle \Box \| \tilde{\mu}x.c \rangle$ and $C' \triangleq \tilde{\mu}y. \langle \mu\alpha.c' \| \Box \rangle$. $C[v_1] = C[v_2]$ follows from $v_1 = v_2$ and $C'[e_1] = C'[e_2]$ follows from $e_1 = e_2$ by just applying the same equalities within the larger context. More interestingly, we can derive $v_1 = v_2$ from $C[v_1] = C[v_2]$ from the definition of the isomorphism by placing them in a larger context, so we have the following equality via $\chi_{\mathcal{S}}$:

$$\begin{aligned}
\mu\alpha. \langle C[v_i] \| \tilde{\mu}y.c' \rangle &\triangleq \mu\alpha. \langle \mu\beta. \langle v_i \| \tilde{\mu}x.c \rangle \| \tilde{\mu}y.c' \rangle \\
&=_{\chi_{\mathcal{S}}} \mu\alpha. \langle v_i \| \tilde{\mu}x. \langle \mu\beta.c \| \tilde{\mu}y.c \rangle \rangle \\
&=_{Iso} \mu\alpha. \langle v_i \| \tilde{\mu}x. \langle x \| \alpha \rangle \rangle =_{\eta_\mu \eta_{\tilde{\mu}}} v_i
\end{aligned}$$

And since we assumed $C[v_1] = C[v_2]$, we have

$$v_1 = \mu\alpha.\, \langle C[v_1] \| \tilde{\mu}y.c' \rangle = \mu\alpha.\, \langle C[v_2] \| \tilde{\mu}y.c' \rangle = v_2$$

$e_1 = e_2$ follows similarly from $C'[e_1] = C'[e_2]$ because of the fact that $\tilde{\mu}x.\, \langle \mu\beta.c \| C[e_i] \rangle = e_i$. □

Finally, we extend the idea of isomorphisms between types to isomorphisms between (co-)data declarations.

Definition 8.2 (Declaration isomorphism)**.** We say that two data declarations are *isomorphic*, written[4]

$$\begin{array}{c}\textbf{data}\,\mathsf{F}(\Theta) : \mathcal{S}\,\textbf{where} \\ \overrightarrow{\mathsf{K} : \left(\Gamma \vdash^{\Theta'} \mathsf{F}(\Theta) \mid \Delta\right)}\end{array} \approx \begin{array}{c}\textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{S}'\,\textbf{where} \\ \overrightarrow{\mathsf{K}' : \left(\Gamma \vdash^{\Theta''} \mathsf{F}'(\Theta) \mid \Delta\right)}\end{array}$$

if and only if $\Theta \vDash \mathsf{F}(\Theta) \approx \mathsf{F}'(\Theta)$. Dually, we say that two co-data declarations are *isomorphic*, written

$$\begin{array}{c}\textbf{codata}\,\mathsf{G}(\Theta) : \mathcal{S}\,\textbf{where} \\ \overrightarrow{\mathsf{O} : \left(\Gamma \mid \mathsf{G}(\Theta) \vdash^{\Theta'} \Delta\right)}\end{array} \approx \begin{array}{c}\textbf{codata}\,\mathsf{G}'(\Theta) : \mathcal{S}'\,\textbf{where} \\ \overrightarrow{\mathsf{O}' : \left(\Gamma' \mid \mathsf{G}'(\Theta) \vdash^{\Theta''} \Delta'\right)}\end{array}$$

if and only if $\Theta \vDash \mathsf{G}(\Theta) \approx \mathsf{G}'(\Theta)$.

Theorem 8.5 (Declaration isomorphism equivalence)**.** *The (co-)data declaration isomorphism relation is (a) reflexive, (b) symmetric, and (c) transitive for any (co-)data types of the same strategy $\mathcal{S}$, such that $\chi_{\mathcal{S}}$ holds.*

Proof. Follows from the reflexivity (Theorem 8.2 (a)), symmetry (Theorem 8.2 (b)), and transitivity (Theorem 8.3) of the type isomorphism relation underlying Definition 8.2. □

This more specific notion of type-based isomorphism is the backbone of the syntactic theory that we will develop for the purpose of reasoning more easily about (co-)data types in general, the polarized basis $\mathcal{P}$ of (co-)data types, and eventually the faithfulness of the polarization translation.

[4] Note that we reuse Γ and Δ as shorthand for the list of types $\overrightarrow{A : \mathcal{T}}$ and $\overrightarrow{B : \mathcal{R}}$ within the signatures of constructors and observers.

A Syntactic Theory of (Co-)Data Type Isomorphisms

Before turning to our main result—that every user-defined (co-)data type can be represented by an isomorphic type composed solely from the polarized basic connectives—we first explore a theory for type isomorphisms based on data and co-data declarations. The advantage of using (co-)data type declarations for studying type isomorphisms is that the declarations themselves provide a larger context for localized manipulations surrounded by extra alternatives (of other constructors and observers) and extra components (within the same constructor or observer). The end result is that we only need to manually verify a few fundamental (co-)data type isomorphisms by hand, while the particular isomorphisms of interest can be easily composed out of these basic building blocks.

Structural laws of declarations

We present an isomorphism theory for the structural laws of data and co-data types in Figures 8.4 and 8.5, which are exactly dual to one another and capture several facts about isomorphic ways to declare (co-)data types.

- *Commute*: The first group of laws state that the parts of any declaration may be reordered, including (1) the order of components within the signature of a constructor or observer, and (2) the order of constructor or observer alternatives within the declaration. These axioms are useful to show that the listed orders of the various parts of a declaration don't matter.

- *Mix*: The second group of laws states how two isomorphism between (co-)data type declarations may be combined together. In particular, there are two ways to mix declaration isomorphisms: (1) an isomorphic pair of single-alternative declarations can have the components of their single constructor or observer mixed into the signature of all the alternatives of another declaration isomorphism to form a larger declaration isomorphism, and (2) a pair of declaration isomorphism can have their respective alternatives mixed together to form a larger isomorphism. These inference rules let us use localized reasoning within a small (co-)data type declaration, and then compose the results together into a large declaration isomorphism that does everything.

- *Shift*: The third group of laws state that every call-by-value ($\mathcal{V}$) data declaration isomorphism and every call-by-name ($\mathcal{N}$) co-data declaration isomorphism may be generalized to (co-)data types of any strategy.
- *Interchange*: The fourth group of laws show how isomorphisms between data type declarations and co-data type declarations can be interchanged one-for-one with one another, so long as the data type is call-by-value ($\mathcal{V}$) and the co-data type is call-by-name ($\mathcal{N}$).
- *Compatibility*: The final group of laws state that an isomorphism between types can be lifted into an isomorphism between data and co-data type declarations with constructors and observations containing a component of that type as either an input or an output.

These laws let us derive other facts about isomorphisms between (co-)data types. As a simple example, applying the shift laws to the trivial cases of the commute laws for data declarations lets us rename constructor and type names, which effectively tells us that there is only one empty and unit type for any strategy $\mathcal{S}$:

$$\begin{gathered}
\mathbf{data}\,\mathsf{F}(\Theta) : \mathcal{S}\,\mathbf{where}\,\mathsf{K} : (\;\vdash \mathsf{F}(\Theta) \mid\;) \approx \mathbf{data}\,\mathsf{F}'(\Theta) : \mathcal{S}\,\mathbf{where}\,\mathsf{K}' : (\;\vdash \mathsf{F}'(\Theta) \mid\;) \\
\mathbf{data}\,\mathsf{F}(\Theta) : \mathcal{S}\,\mathbf{where} \approx \mathbf{data}\,\mathsf{F}'(\Theta) : \mathcal{S}\,\mathbf{where}
\end{gathered}$$

Additionally, the mix laws let us extend an existing isomorphism by combining it with a reflexive isomorphism of any declaration, letting us add on arbitrary other alternatives or components to two isomorphic data declarations:

$$\frac{
\begin{array}{l}
\mathbf{data}\,\mathsf{F}_1(\Theta) : \mathcal{V}\,\mathbf{where} \\
\quad \overrightarrow{\mathsf{K}_1 : (\Gamma_1 \vdash \mathsf{F}(\Theta) \mid \Delta_1)} \\
\approx \begin{array}{l} \mathbf{data}\,\mathsf{F}_1'(\Theta) : \mathcal{V}\,\mathbf{where} \\ \quad \overrightarrow{\mathsf{K}_1' : (\Gamma_1' \vdash \mathsf{F}'(\Theta) \mid \Delta_1')} \end{array}
\end{array}
\qquad
\dfrac{}{
\begin{array}{l}
\mathbf{data}\,\mathsf{F}_2(\Theta) : \mathcal{V}\,\mathbf{where} \\
\quad \overrightarrow{\mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}(\Theta) \mid \Delta_2)} \\
\approx \begin{array}{l} \mathbf{data}\,\mathsf{F}_2(\Theta) : \mathcal{V}\,\mathbf{where} \\ \quad \overrightarrow{\mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}(\Theta) \mid \Delta_2)} \end{array}
\end{array}}
}{
\begin{array}{l}
\mathbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\mathbf{where} \\
\quad \overrightarrow{\mathsf{K}_1 : (\Gamma_1 \vdash \mathsf{F}(\Theta) \mid \Delta_1)} \\
\quad \overrightarrow{\mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}(\Theta) \mid \Delta_2)}
\end{array}
\approx
\begin{array}{l}
\mathbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\mathbf{where} \\
\quad \overrightarrow{\mathsf{K}_1' : (\Gamma_1' \vdash \mathsf{F}'(\Theta) \mid \Delta_1')} \\
\quad \overrightarrow{\mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}(\Theta) \mid \Delta_2)}
\end{array}
}$$

$$\dfrac{\begin{array}{c}\mathbf{data}\,\mathsf{F}_1(\Theta):\mathcal{V}\,\mathbf{where}\\ \overrightarrow{\mathsf{K}_1 : (\Gamma_1 \vdash \mathsf{F}(\Theta) \mid \Delta_1)}\end{array} \approx \begin{array}{c}\mathbf{data}\,\mathsf{F}'_1(\Theta):\mathcal{V}\,\mathbf{where}\\ \overrightarrow{\mathsf{K}'_1 : (\Gamma'_1 \vdash \mathsf{F}'(\Theta) \mid \Delta'_1)}\end{array} \qquad \dfrac{}{\begin{array}{c}\mathbf{data}\,\mathsf{F}_2(\Theta):\mathcal{V}\,\mathbf{where}\\ \mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}(\Theta) \mid \Delta_2)\end{array} \approx \begin{array}{c}\mathbf{data}\,\mathsf{F}_2(\Theta):\mathcal{V}\,\mathbf{where}\\ \mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}(\Theta) \mid \Delta_2)\end{array}}}{\begin{array}{c}\mathbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\mathbf{where}\\ \overrightarrow{\mathsf{K}_1 : (\Gamma_2, \Gamma_1 \vdash \mathsf{F}(\Theta) \mid \Delta_1, \Delta_1)}\end{array} \approx \begin{array}{c}\mathbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\mathbf{where}\\ \overrightarrow{\mathsf{K}'_1 : (\Gamma_2, \Gamma'_1 \vdash \mathsf{F}'(\Theta) \mid \Delta'_1, \Delta_2)}\end{array}}$$

We can justify the laws in Figures 8.4 and 8.5 in terms of the definitions of type and (co-)data declaration isomorphisms. In particular, we can calculate when specific instances of two (co-)data types happen to be isomorphic, so that the laws for declaration isomorphisms are sound when the specific instances hold in general for any matching choice of types. These calculations can be done for the commute, mix, compatibility, interchange and shift laws for data declarations as follows.

Lemma 8.1 (Data commute instance)**.** *For any types $\overrightarrow{C : \mathcal{S}}$, $\overrightarrow{C' : \mathcal{S}'}$, $\mathsf{F}(\overrightarrow{C}) \approx \mathsf{F}'(\overrightarrow{C'})$ for the declarations*

a)
$$\begin{array}{c}\mathbf{data}\,\mathsf{F}(\overrightarrow{X : \mathcal{S}}) : \mathcal{V}\,\mathbf{where}\\ \mathsf{K}\colon \left(\Gamma_2, \Gamma_1 \vdash \mathsf{F}(\overrightarrow{X}) \mid \Delta_1, \Delta_2\right)\end{array} \quad \textit{and} \quad \begin{array}{c}\mathbf{data}\,\mathsf{F}'(\overrightarrow{X' : \mathcal{S}'}) : \mathcal{V}\,\mathbf{where}\\ \mathsf{K}'\colon \left(\Gamma'_1, \Gamma'_2 \vdash \mathsf{F}'(\overrightarrow{X'}) \mid \Delta'_2, \Delta'_1\right)\end{array}$$
such that $\Gamma_1\theta = \Gamma'_1\theta'$, $\Gamma_2\theta = \Gamma'_2\theta'$, $\Delta_1\theta = \Delta'_1\theta'$, and $\Delta_1\theta = \Delta'_1\theta'$ where $\theta = \overrightarrow{\{C/X\}}$ and $\theta' = \overrightarrow{\{C'/X'\}}$

b)
$$\begin{array}{c}\mathbf{data}\,\mathsf{F}(\overrightarrow{X : \mathcal{S}}) : \mathcal{V}\,\mathbf{where}\\ \overrightarrow{\mathsf{K}_1 : \left(\Gamma_1 \vdash \mathsf{F}(\overrightarrow{X}) \mid \Delta_1\right)}\\ \overrightarrow{\mathsf{K}_2 : \left(\Gamma_2 \vdash \mathsf{F}(\overrightarrow{X}) \mid \Delta_2\right)}\end{array} \quad \textit{and} \quad \begin{array}{c}\mathbf{data}\,\mathsf{F}'(\overrightarrow{X' : \mathcal{S}'}) : \mathcal{V}\,\mathbf{where}\\ \overrightarrow{\mathsf{K}'_2 : \left(\Gamma'_2 \vdash \mathsf{F}'(\overrightarrow{X'}) \mid \Delta'_2\right)}\\ \overrightarrow{\mathsf{K}'_1 : \left(\Gamma'_1 \vdash \mathsf{F}'(\overrightarrow{X'}) \mid \Delta'_1\right)}\end{array}$$
such that $\overrightarrow{\Gamma_1\theta = \Gamma'_1\theta'}$, $\overrightarrow{\Gamma_2\theta = \Gamma'_2\theta'}$, $\overrightarrow{\Delta_1\theta = \Delta'_1\theta'}$, and $\overrightarrow{\Delta_2\theta = \Delta'_2\theta'}$ where $\theta = \overrightarrow{\{C/X\}}$ and $\theta' = \overrightarrow{\{C'/X'\}}$

Proof. The isomorphisms between $\mathsf{F}(\overrightarrow{C})$ and $\mathsf{F}'(\overrightarrow{C'})$ are established by the commands $c : (x : \mathsf{F}(\overrightarrow{C}) \vdash \alpha' : \mathsf{F}'(\overrightarrow{C'}))$ and $c' : (x' : \mathsf{F}'(\overrightarrow{C'}) \vdash \alpha : \mathsf{F}(\overrightarrow{C}))$ as follows:

a)
$$\begin{aligned} c &\triangleq \langle x \| \tilde{\mu}[\mathsf{K}(\overrightarrow{\beta_1}, \overrightarrow{\beta_2}, \overrightarrow{y_2}, \overrightarrow{y_1}).\langle \mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}) \| \alpha' \rangle] \rangle \\ c' &\triangleq \langle x' \| \tilde{\mu}[\mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}).\langle \mathsf{K}(\overrightarrow{\beta_1}, \overrightarrow{\beta_2}, \overrightarrow{y_2}, \overrightarrow{y_1}) \| \alpha \rangle] \rangle \end{aligned}$$

b)
$$c \triangleq \left\langle x \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}'_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha' \rangle \\ \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}'_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha' \rangle \end{bmatrix} \right\rangle \quad c' \triangleq \left\langle x' \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}'_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha \rangle \\ \mathsf{K}'_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha \rangle \end{bmatrix} \right\rangle$$

Data Commute

$$\begin{array}{l}\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\mathsf{K} : (\Gamma_2,\Gamma_1 \vdash \mathsf{F}(\Theta)|\Delta_1,\Delta_2)\end{array} \approx \begin{array}{l}\textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\mathsf{K}' : (\Gamma_1,\Gamma_2 \vdash \mathsf{F}'(\Theta)|\Delta_2,\Delta_1)\end{array}$$

$$\begin{array}{l}\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{K}_1 : (\Gamma_1 \vdash \mathsf{F}(\Theta)|\Delta_1)}\\ \quad\overrightarrow{\mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}(\Theta)|\Delta_2)}\end{array} \approx \begin{array}{l}\textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{K}'_2 : (\Gamma_2 \vdash \mathsf{F}'(\Theta)|\Delta_2)}\\ \quad\overrightarrow{\mathsf{K}'_1 : (\Gamma_1 \vdash \mathsf{F}'(\Theta)|\Delta_1)}\end{array}$$

Data Mix

$$\frac{\begin{array}{l}\textbf{data}\,\mathsf{F}_1(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{K}_1 : (\Gamma_1 \vdash \mathsf{F}_1(\Theta)|\Delta_1)}\\ \approx\;\textbf{data}\,\mathsf{F}'_1(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\quad\overrightarrow{\mathsf{K}'_1 : (\Gamma'_1 \vdash \mathsf{F}'_1(\Theta)|\Delta'_1)}\end{array} \qquad \begin{array}{l}\textbf{data}\,\mathsf{F}_2(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}_2(\Theta)|\Delta_2)\\ \approx\;\textbf{data}\,\mathsf{F}'_2(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\quad\mathsf{K}'_2 : (\Gamma'_2 \vdash \mathsf{F}'_2(\Theta)|\Delta'_2)\end{array}}{\begin{array}{l}\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{K} : (\Gamma_2,\Gamma_1 \vdash \mathsf{F}(\Theta)|\Delta_1,\Delta_2)}\end{array} \approx \begin{array}{l}\textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{K}' : (\Gamma'_2,\Gamma'_1 \vdash \mathsf{F}'(\Theta)|\Delta'_1,\Delta'_2)}\end{array}}$$

$$\frac{\begin{array}{l}\textbf{data}\,\mathsf{F}_1(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{K}_1 : (\Gamma_1 \vdash \mathsf{F}_1(\Theta)|\Delta_1)}\\ \approx\;\textbf{data}\,\mathsf{F}'_1(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\quad\overrightarrow{\mathsf{K}'_1 : (\Gamma'_1 \vdash \mathsf{F}'_1(\Theta)|\Delta'_1)}\end{array} \qquad \begin{array}{l}\textbf{data}\,\mathsf{F}_2(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}_2(\Theta)|\Delta_2)}\\ \approx\;\textbf{data}\,\mathsf{F}'_2(\Theta):\mathcal{V}\,\textbf{where}\\ \quad\quad\overrightarrow{\mathsf{K}'_2 : (\Gamma'_2 \vdash \mathsf{F}'_2(\Theta)|\Delta'_2)}\end{array}}{\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\begin{array}{l}\overrightarrow{\mathsf{K}_1 : (\Gamma_1 \vdash \mathsf{F}(\Theta)|\Delta_1)}\\ \overrightarrow{\mathsf{K}_2 : (\Gamma_2 \vdash \mathsf{F}(\Theta)|\Delta_2)}\end{array} \approx \textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\begin{array}{l}\overrightarrow{\mathsf{K}'_1 : (\Gamma'_1 \vdash \mathsf{F}'(\Theta)|\Delta'_1)}\\ \overrightarrow{\mathsf{K}'_2 : (\Gamma'_2 \vdash \mathsf{F}'(\Theta)|\Delta'_2)}\end{array}}$$

Data Shift

$$\frac{\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K} : (\Gamma \vdash \mathsf{F}(\Theta)|\Delta)} \approx \textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K}' : (\Gamma' \vdash \mathsf{F}'(\Theta)|\Delta')}}{\textbf{data}\,\mathsf{F}(\Theta):\mathcal{S}\,\textbf{where}\,\overrightarrow{\mathsf{K} : (\Gamma \vdash \mathsf{F}(\Theta)|\Delta)} \approx \textbf{data}\,\mathsf{F}'(\Theta):\mathcal{S}'\,\textbf{where}\,\overrightarrow{\mathsf{K}' : (\Gamma' \vdash \mathsf{F}'(\Theta)|\Delta')}}$$

Co-data-Data Interchange

$$\frac{\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\,\overrightarrow{\mathsf{O} : \left(\Gamma|\mathsf{G}(\overrightarrow{X}) \vdash \Delta\right)} \approx \textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{N}\,\textbf{where}\,\overrightarrow{\mathsf{O}' : \left(\Gamma'|\mathsf{G}'(\overrightarrow{X'}) \vdash \Delta'\right)}}{\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K} : (\Gamma \vdash \mathsf{F}(\Theta)|\Delta)} \approx \textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K}' : \left(\Gamma' \vdash \mathsf{F}'(\overrightarrow{X'})|\Delta'\right)}}$$

Data Compatibility

$$\frac{\Theta \vdash A : \mathcal{S} \quad \Theta \vdash B : \mathcal{S} \quad \Theta \vDash A \approx B}{\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\,\mathsf{K} : (A : \mathcal{S} \vdash \mathsf{F}(\Theta)|) \approx \textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\,\mathsf{K}' : (B : \mathcal{S} \vdash \mathsf{F}'(\Theta)|)}$$

$$\frac{\Theta \vdash A : \mathcal{S} \quad \Theta \vdash B : \mathcal{S} \quad \Theta \vDash A \approx B}{\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\,\mathsf{K} : (\vdash \mathsf{F}(\Theta)|A : \mathcal{S}) \approx \textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\,\mathsf{K}' : (\vdash \mathsf{F}'(\Theta)|B : \mathcal{S})}$$

FIGURE 8.4. A theory for structural laws of data type declaration isomorphisms.

Co-data Commute

$$\begin{array}{l}\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\\ \quad \mathsf{O} : (\Gamma_2,\Gamma_1|\mathsf{G}(\Theta)\vdash\Delta_1,\Delta_2)\end{array} \approx \begin{array}{l}\textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{N}\,\textbf{where}\\ \quad \mathsf{O}' : (\Gamma_1,\Gamma_2|\mathsf{G}'(\Theta)\vdash\Delta_2,\Delta_1)\end{array}$$

$$\begin{array}{r}\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\\ \overrightarrow{\mathsf{O}_1 : (\Gamma_1|\mathsf{G}(\Theta)\vdash\Delta_1)}\\ \overrightarrow{\mathsf{O}_2 : (\Gamma_2|\mathsf{G}(\Theta)\vdash\Delta_2)}\end{array} \approx \begin{array}{r}\textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{S}\,\textbf{where}\\ \overrightarrow{\mathsf{O}_2' : (\Gamma_2|\mathsf{G}'(\Theta)\vdash\Delta_2)}\\ \overrightarrow{\mathsf{O}_1' : (\Gamma_1|\mathsf{G}'(\Theta)\vdash\Delta_1)}\end{array}$$

Co-data Mix

$$\frac{\begin{array}{l}\textbf{codata}\,\mathsf{G}_1(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{O}_1 : (\Gamma_1|\mathsf{G}_1(\Theta)\vdash\Delta_1)}\\ \approx \begin{array}{l}\textbf{codata}\,\mathsf{G}_1'(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{O}_1' : (\Gamma_1'|\mathsf{G}_1'(\Theta)\vdash\Delta_1')}\end{array}\end{array} \qquad \begin{array}{l}\textbf{codata}\,\mathsf{G}_2(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\mathsf{O}_2 : (\Gamma_2|\mathsf{G}_2(\Theta)\vdash\Delta_2)\\ \approx \begin{array}{l}\textbf{data}\,\mathsf{G}_2'(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\mathsf{O}_2' : (\Gamma_2'|\mathsf{G}_2'(\Theta)\vdash\Delta_2')\end{array}\end{array}}{\begin{array}{l}\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{O} : (\Gamma_2,\Gamma_1|\mathsf{G}(\Theta)\vdash\Delta_1,\Delta_2)}\end{array} \approx \begin{array}{l}\textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{O}' : (\Gamma_2',\Gamma_1'|\mathsf{G}'(\Theta)\vdash\Delta_1',\Delta_2')}\end{array}}$$

$$\frac{\begin{array}{l}\textbf{codata}\,\mathsf{G}_1(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{O}_1 : (\Gamma_1|\mathsf{G}_1(\Theta)\vdash\Delta_1)}\\ \approx \begin{array}{l}\textbf{codata}\,\mathsf{G}_1'(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\overrightarrow{\mathsf{O}_1' : (\Gamma_1'|\mathsf{G}_1'(\Theta)\vdash\Delta_1')}\end{array}\end{array} \qquad \begin{array}{l}\textbf{codata}\,\mathsf{G}_2(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\mathsf{O}_2 : (\Gamma_2|\mathsf{G}_2(\Theta)\vdash\Delta_2)\\ \approx \begin{array}{l}\textbf{data}\,\mathsf{G}_2'(\Theta):\mathcal{N}\,\textbf{where}\\ \quad\mathsf{O}_2' : (\Gamma_2'|\mathsf{G}_2'(\Theta)\vdash\Delta_2')\end{array}\end{array}}{\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\begin{array}{l}\overrightarrow{\mathsf{O}_1 : (\Gamma_1|\mathsf{G}(\Theta)\vdash\Delta_1)}\\ \overrightarrow{\mathsf{O}_2 : (\Gamma_2|\mathsf{G}(\Theta)\vdash\Delta_2)}\end{array} \approx \textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{N}\,\textbf{where}\begin{array}{l}\overrightarrow{\mathsf{O}_1' : (\Gamma_1'|\mathsf{G}'(\Theta)\vdash\Delta_1')}\\ \overrightarrow{\mathsf{O}_2' : (\Gamma_2'|\mathsf{G}'(\Theta)\vdash\Delta_2')}\end{array}}$$

Co-data Shift

$$\frac{\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\,\overrightarrow{\mathsf{O} : (\Gamma|\mathsf{G}(\Theta)\vdash\Delta)} \approx \textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{N}\,\textbf{where}\,\overrightarrow{\mathsf{O}' : (\Gamma'|\mathsf{G}'(\Theta)\vdash\Delta')}}{\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{S}\,\textbf{where}\,\overrightarrow{\mathsf{O} : (\Gamma|\mathsf{G}(\Theta)\vdash\Delta)} \approx \textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{S}'\,\textbf{where}\,\overrightarrow{\mathsf{O}' : (\Gamma'|\mathsf{G}'(\Theta)\vdash\Delta')}}$$

Data-Co-data Interchange

$$\frac{\textbf{data}\,\mathsf{F}(\Theta):\mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K} : (\Gamma\vdash\mathsf{F}(\Theta)|\Delta)} \approx \textbf{data}\,\mathsf{F}'(\Theta):\mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K}' : \left(\Gamma'\vdash\mathsf{F}'(\overrightarrow{X'})|\Delta'\right)}}{\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\,\overrightarrow{\mathsf{O} : \left(\Gamma|\mathsf{G}(\overrightarrow{X})\vdash\Delta\right)} \approx \textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{N}\,\textbf{where}\,\overrightarrow{\mathsf{O}' : \left(\Gamma'|\mathsf{G}'(\overrightarrow{X'})\vdash\Delta'\right)}}$$

Co-data Compatibility

$$\frac{\Theta\vdash A:\mathcal{S}\quad \Theta\vdash B:\mathcal{S}\quad \Theta\vDash A\approx B}{\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\,\mathsf{O} : (|\mathsf{G}(\Theta)\vdash A:\mathcal{S}) \approx \textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{N}\,\textbf{where}\,\mathsf{O}' : (|\mathsf{G}'(\Theta)\vdash B:\mathcal{S})}$$

$$\frac{\Theta\vdash A:\mathcal{S}\quad \Theta\vdash B:\mathcal{S}\quad \Theta\vDash A\approx B}{\textbf{codata}\,\mathsf{G}(\Theta):\mathcal{N}\,\textbf{where}\,\mathsf{O} : (A:\mathcal{S}|\mathsf{G}(\Theta)\vdash\,) \approx \textbf{codata}\,\mathsf{G}'(\Theta):\mathcal{N}\,\textbf{where}\,\mathsf{O}' : (B:\mathcal{S}|\mathsf{G}'(\Theta)\vdash\,)}$$

FIGURE 8.5. A theory for structural laws of co-data type declaration isomorphisms.

For part (a), the composition of c' and c along α and x of type $\mathsf{F}(\overrightarrow{C})$ is equal to the identity command $\langle x' \| \alpha' \rangle$ via the β^{F} and $\eta^{\mathsf{F}'}$ axioms as follows:

$$
\begin{aligned}
&\langle \mu\alpha.c' \| \tilde{\mu}x.c \rangle \\
&\triangleq \Big\langle \mu\alpha.\langle x' \| \tilde{\mu}[\mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}).\langle \mathsf{K}(\overrightarrow{\beta_1}, \overrightarrow{\beta_2}, \overrightarrow{y_2}, \overrightarrow{y_1}) \| \alpha \rangle] \rangle \\
&\qquad \Big\| \mu x.\langle x \| \tilde{\mu}[\mathsf{K}(\overrightarrow{\beta_1}, \overrightarrow{\beta_2}, \overrightarrow{y_2}, \overrightarrow{y_1}).\langle \mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}) \| \alpha' \rangle] \rangle \Big\rangle \\
&=_{\eta_{\tilde{\mu}}} \Big\langle \mu\alpha.\langle x' \| \tilde{\mu}[\mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}).\langle \mathsf{K}(\overrightarrow{\beta_1}, \overrightarrow{\beta_2}, \overrightarrow{y_2}, \overrightarrow{y_1}) \| \alpha \rangle] \rangle \\
&\qquad\quad \Big\| \tilde{\mu}[\mathsf{K}(\overrightarrow{\beta_1}, \overrightarrow{\beta_2}, \overrightarrow{y_2}, \overrightarrow{y_1}).\langle \mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}) \| \alpha' \rangle] \Big\rangle \\
&=_{\mu_{\mathcal{V}}} \Big\langle x' \Big\| \tilde{\mu}\Big[\mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}).\langle \mathsf{K}(\overrightarrow{\beta_1}, \overrightarrow{\beta_2}, \overrightarrow{y_2}, \overrightarrow{y_1}) \| \tilde{\mu}[\mathsf{K}(\overrightarrow{\beta_1}, \overrightarrow{\beta_2}, \overrightarrow{y_2}, \overrightarrow{y_1}).\langle \mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}) \| \alpha' \rangle] \rangle \Big] \Big\rangle \\
&=_{\beta^{\mathsf{F}} \mu_\alpha \tilde{\mu}_x} \langle x' \| \tilde{\mu}[\mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}).\langle \mathsf{K}'(\overrightarrow{\beta_2}, \overrightarrow{\beta_1}, \overrightarrow{y_1}, \overrightarrow{y_2}) \| \alpha' \rangle] \rangle \\
&=_{\eta^{\mathsf{F}'}} \langle x' \| \alpha' \rangle
\end{aligned}
$$

And the reverse composition of c and c' along α' and x' of type $\mathsf{F}'(\overrightarrow{C'})$ is similarly equal to the identity command $\langle x \| \alpha \rangle$ via the $\beta^{\mathsf{F}'}$ and η^{F}.

For part (b), the composition of c' and c along α and x of type $\mathsf{F}(\overrightarrow{C})$ is equal to the identity command $\langle x' \| \alpha' \rangle$ via the β^{F} and $\eta^{\mathsf{F}'}$ axioms as follows:

$$
\begin{aligned}
&\langle \mu\alpha.c' \| \tilde{\mu}x.c \rangle \\
&\triangleq \left\langle \mu\alpha. \left\langle x' \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha \rangle \\ \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha \rangle \end{bmatrix} \right\rangle \right. \\
&\qquad \left. \middle\| \tilde{\mu}x. \left\langle x \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha' \rangle \\ \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha' \rangle \end{bmatrix} \right\rangle \right\rangle \\
&=_{\eta_{\tilde{\mu}}} \left\langle \mu\alpha. \left\langle x' \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha \rangle \\ \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha \rangle \end{bmatrix} \right\rangle \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha' \rangle \\ \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha' \rangle \end{bmatrix} \right\rangle \\
&=_{\mu_{\mathcal{V}}} \left\langle x' \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}). \left\langle \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha' \rangle \\ \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha' \rangle \end{bmatrix} \right\rangle \\ \\ \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}). \left\langle \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_1(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha' \rangle \\ \mathsf{K}_2(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha' \rangle \end{bmatrix} \right\rangle \end{bmatrix} \right\rangle \\
&=_{\beta^{\mathsf{F}} \mu_\alpha \tilde{\mu}_x} \left\langle x' \middle\| \tilde{\mu} \begin{bmatrix} \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}).\langle \mathsf{K}_2'(\overrightarrow{\beta_2}, \overrightarrow{y_2}) \| \alpha' \rangle \\ \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}).\langle \mathsf{K}_1'(\overrightarrow{\beta_1}, \overrightarrow{y_1}) \| \alpha' \rangle \end{bmatrix} \right\rangle =_{\eta^{\mathsf{F}'}} \langle x' \| \alpha' \rangle
\end{aligned}
$$

And the reverse composition of c and c' along α' and x' of type $\mathsf{F}'(\overrightarrow{C'})$ is equal to the identity command $\langle x \| \alpha \rangle$ via the $\beta^{\mathsf{F}'}$ and η^{F}. □

Lemma 8.2 (Data mix instance)**.** *For any data types declared as*

$$\begin{array}{ll}
\textbf{data}\,\mathsf{F}_1(\overrightarrow{X : \mathcal{S}}) : \mathcal{V}\,\textbf{where} & \textbf{data}\,\mathsf{F}'_1(\overrightarrow{X' : \mathcal{S}'}) : \mathcal{V}\,\textbf{where} \\
\quad \overrightarrow{\mathsf{K}_1 : \left(\Gamma_1 \vdash \mathsf{F}_1(\overrightarrow{X}) \mid \Delta_1\right)} & \quad \overrightarrow{\mathsf{K}'_1 : \left(\Gamma'_1 \vdash \mathsf{F}'_1(\overrightarrow{X'}) \mid \Delta'_1\right)} \\
\textbf{data}\,\mathsf{F}_2(\overrightarrow{X : \mathcal{S}}) : \mathcal{V}\,\textbf{where} & \textbf{data}\,\mathsf{F}'_2(\overrightarrow{X' : \mathcal{S}'}) : \mathcal{V}\,\textbf{where} \\
\quad \overrightarrow{\mathsf{K}_2 : \left(\Gamma_2 \vdash \mathsf{F}_2(\overrightarrow{X}) \mid \Delta_2\right)} & \quad \overrightarrow{\mathsf{K}'_2 : \left(\Gamma'_2 \vdash \mathsf{F}'_2(\overrightarrow{X'}) \mid \Delta'_2\right)} \\
\textbf{data}\,\mathsf{F}_3(\overrightarrow{X : \mathcal{S}}) : \mathcal{V}\,\textbf{where} & \textbf{data}\,\mathsf{F}'_3(\overrightarrow{X' : \mathcal{S}'}) : \mathcal{V}\,\textbf{where} \\
\quad \mathsf{K}_3 : \left(\Gamma_3 \vdash \mathsf{F}_3(\overrightarrow{X}) \mid \Delta_3\right) & \quad \mathsf{K}'_3 : \left(\Gamma'_3 \vdash \mathsf{F}'_3(\overrightarrow{X'}) \mid \Delta'_3\right) \\
\textbf{data}\,\mathsf{F}(\overrightarrow{X : \mathcal{S}}) : \mathcal{V}\,\textbf{where} & \textbf{data}\,\mathsf{F}'(\overrightarrow{X' : \mathcal{S}}) : \mathcal{V}\,\textbf{where} \\
\quad \overrightarrow{\mathsf{K}_4 : \left(\Gamma_3, \Gamma_1 \vdash \mathsf{F}(\overrightarrow{X}) \mid \Delta_1, \Delta_3\right)} & \quad \overrightarrow{\mathsf{K}'_4 : \left(\Gamma'_3, \Gamma'_1 \vdash \mathsf{F}'(\overrightarrow{X'}) \mid \Delta'_1, \Delta'_3\right)} \\
\quad \overrightarrow{\mathsf{K}_5 : \left(\Gamma_3, \Gamma_2 \vdash \mathsf{F}(\overrightarrow{X}) \mid \Delta_2, \Delta_3\right)} & \quad \overrightarrow{\mathsf{K}'_5 : \left(\Gamma'_3, \Gamma'_2 \vdash \mathsf{F}'(\overrightarrow{X'}) \mid \Delta'_2, \Delta'_3\right)}
\end{array}$$

and types $\overrightarrow{C : \mathcal{S}}$*,* $\overrightarrow{C' : \mathcal{S}'}$*, if* $\mathsf{F}_1(\overrightarrow{C}) \approx \mathsf{F}'_1(\overrightarrow{C'})$*,* $\mathsf{F}_2(\overrightarrow{C}) \approx \mathsf{F}'_2(\overrightarrow{C'})$*, and* $\mathsf{F}_3(\overrightarrow{C}) \approx \mathsf{F}'_3(\overrightarrow{C'})$*, then* $\mathsf{F}(\overrightarrow{C}) \approx \mathsf{F}'(\overrightarrow{C'})$*.*

Lemma 8.3 (Data compatibility instance)**.** *For types* $A : \mathcal{T}$*,* $A' : \mathcal{T}$*,* $\overrightarrow{C : \mathcal{S}}$*,* $\overrightarrow{C' : \mathcal{S}'}$*, if* $A\overrightarrow{\{C/X\}} \approx A'\overrightarrow{\{C'/X'\}}$ *then* $\mathsf{F}(\overrightarrow{C}) \approx \mathsf{F}'(\overrightarrow{C'})$ *for the following declarations:*

a)
$$\begin{array}{c} \textbf{data}\,\mathsf{F}(\overrightarrow{X : \mathcal{S}}) : \mathcal{V}\,\textbf{where} \\ \mathsf{K} : \left(A : \mathcal{T} \vdash \mathsf{F}(\overrightarrow{X}) \mid \right) \end{array} \quad \textit{and} \quad \begin{array}{c} \textbf{data}\,\mathsf{F}'(\overrightarrow{X' : \mathcal{S}'}) : \mathcal{V}\,\textbf{where} \\ \mathsf{K}' : \left(A' : \mathcal{T} \vdash \mathsf{F}'(\overrightarrow{X'}) \mid \right) \end{array}$$

b)
$$\begin{array}{c} \textbf{data}\,\mathsf{F}(\overrightarrow{X : \mathcal{S}}) : \mathcal{V}\,\textbf{where} \\ \mathsf{K} : \left(\vdash \mathsf{F}(\overrightarrow{X}) \mid A : \mathcal{T}\right) \end{array} \quad \textit{and} \quad \begin{array}{c} \textbf{data}\,\mathsf{F}'(\overrightarrow{X' : \mathcal{S}'}) : \mathcal{V}\,\textbf{where} \\ \mathsf{K}' : \left(\vdash \mathsf{F}'(\overrightarrow{X'}) \mid A' : \mathcal{T}\right) \end{array}$$

Proof. Suppose that the isomorphisms $\mathsf{F}_1(\overrightarrow{C}) \approx \mathsf{F}'_1(\overrightarrow{C'})$, $\mathsf{F}_2(\overrightarrow{C}) \approx \mathsf{F}'_2(\overrightarrow{C'})$, and $\mathsf{F}_3(\overrightarrow{C}) \approx \mathsf{F}'_3(\overrightarrow{C'})$ are witnessed by the commands

$$\begin{array}{ll}
c_1 : (x_1 : \mathsf{F}_1(\overrightarrow{C}) \vdash \alpha'_1 : \mathsf{F}'_1(\overrightarrow{C'})) & c'_1 : (x'_1 : \mathsf{F}'_1(\overrightarrow{C'}) \vdash \alpha_1 : \mathsf{F}_1(\overrightarrow{C})) \\
c_2 : (x_2 : \mathsf{F}_2(\overrightarrow{C}) \vdash \alpha'_2 : \mathsf{F}'_2(\overrightarrow{C'})) & c'_2 : (x'_2 : \mathsf{F}'_2(\overrightarrow{C'}) \vdash \alpha_2 : \mathsf{F}_2(\overrightarrow{C})) \\
c_3 : (x_3 : \mathsf{F}_3(\overrightarrow{C}) \vdash \alpha'_3 : \mathsf{F}'_3(\overrightarrow{C'})) & c'_3 : (x'_3 : \mathsf{F}'_3(\overrightarrow{C'}) \vdash \alpha_3 : \mathsf{F}_3(\overrightarrow{C}))
\end{array}$$

respectively. Then isomorphisms between $\mathsf{F}(\overrightarrow{C})$ and $\mathsf{F}'(\overrightarrow{C'})$ are established by the commands $c : (x : \mathsf{F}(\overrightarrow{C}) \vdash \alpha' : \mathsf{F}'(\overrightarrow{C'}))$ and $c' : (x' : \mathsf{F}'(\overrightarrow{C'}) \vdash \alpha : \mathsf{F}(\overrightarrow{C}))$ as follows:

$$c \triangleq \left\langle x \middle\| \tilde{\mu} \left[\begin{array}{l} \overrightarrow{\begin{array}{l} \mathsf{K}_{4i}(\overrightarrow{\beta_{1i}}, \overrightarrow{\beta_3}, \overrightarrow{y_3}, \overrightarrow{y_{1i}}). \\ \quad \left\langle v'_{1i} \middle\| \tilde{\mu} \left[\overrightarrow{\mathsf{K}'_{1i}(\overrightarrow{\beta'_{1j}}, \overrightarrow{y'_{1j}}). \left\langle v'_3 \middle\| \tilde{\mu} \left[\mathsf{K}'_3(\overrightarrow{\beta'_3}, \overrightarrow{y'_3}). \left\langle \mathsf{K}'_{4j}(\overrightarrow{\beta'_{1j}, \overrightarrow{\beta'_3}, \overrightarrow{y'_3}, \overrightarrow{y'_{1j}}}) \middle\| \alpha' \right\rangle \right] \right\rangle}^{j} \right] \right\rangle \end{array}}^{i} \\ \overrightarrow{\begin{array}{l} \mathsf{K}_{5i}(\overrightarrow{\beta_{2i}}, \overrightarrow{\beta_3}, \overrightarrow{y_3}, \overrightarrow{y_{2i}}). \\ \quad \left\langle v'_{2i} \middle\| \tilde{\mu} \left[\overrightarrow{\mathsf{K}'_{2i}(\overrightarrow{\beta'_{2j}}, \overrightarrow{y'_{2j}}). \left\langle v'_3 \middle\| \tilde{\mu} \left[\mathsf{K}'_3(\overrightarrow{\beta'_3}, \overrightarrow{y'_3}). \left\langle \mathsf{K}'_{5j}(\overrightarrow{\beta'_{2j}, \overrightarrow{\beta'_3}, \overrightarrow{y'_3}, \overrightarrow{y'_{2j}}}) \middle\| \alpha' \right\rangle \right] \right\rangle}^{j} \right] \right\rangle \end{array}}^{i} \end{array} \right] \right\rangle$$

$$c' \triangleq \left\langle x' \middle\| \tilde{\mu} \left[\begin{array}{l} \overrightarrow{\begin{array}{l} \mathsf{K}'_{4i}(\overrightarrow{\beta'_{1i}}, \overrightarrow{\beta'_3}, \overrightarrow{y'_3}, \overrightarrow{y'_{1i}}). \\ \quad \left\langle v_3 \middle\| \tilde{\mu} \left[\mathsf{K}_3(\overrightarrow{\beta_3}, \overrightarrow{y_3}). \left\langle v_{1i} \middle\| \tilde{\mu} \left[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1j}}, \overrightarrow{y_{1j}}). \left\langle \mathsf{K}_{4j}(\overrightarrow{\beta_{1j}}, \overrightarrow{\beta_3}, \overrightarrow{y_3}, \overrightarrow{\beta_{1j}}) \middle\| \alpha \right\rangle}^{j} \right] \right\rangle \right] \right\rangle \end{array}}^{i} \\ \overrightarrow{\begin{array}{l} \mathsf{K}'_{5i}(\overrightarrow{\beta'_{2i}}, \overrightarrow{\beta'_3}, \overrightarrow{y'_3}, \overrightarrow{y'_{2i}}). \\ \quad \left\langle v_3 \middle\| \tilde{\mu} \left[\mathsf{K}_3(\overrightarrow{\beta_3}, \overrightarrow{y_3}). \left\langle v_{2i} \middle\| \tilde{\mu} \left[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2j}}, \overrightarrow{y_{2j}}). \left\langle \mathsf{K}_{5j}(\overrightarrow{\beta_{1j}}, \overrightarrow{\beta_3}, \overrightarrow{y_3}, \overrightarrow{\beta_{1j}}) \middle\| \alpha \right\rangle}^{j} \right] \right\rangle \right] \right\rangle \end{array}}^{i} \end{array} \right] \right\rangle$$

where we make use of the following shorthand:

$$\begin{aligned} v_{1i} &\triangleq \mu\alpha_1. \left\langle \mathsf{K}'_{1i}(\overrightarrow{\beta'_{1i}}, \overrightarrow{y'_{1i}}) \middle\| \tilde{\mu}x'_1.c'_1 \right\rangle & v'_{1i} &\triangleq \mu\alpha'_1. \left\langle \mathsf{K}_{1i}(\overrightarrow{\beta_{1i}}, \overrightarrow{y_{1i}}) \middle\| \tilde{\mu}x_1.c_1 \right\rangle \\ v_{2i} &\triangleq \mu\alpha_2. \left\langle \mathsf{K}'_{2i}(\overrightarrow{\beta'_{2i}}, \overrightarrow{y'_{2i}}) \middle\| \tilde{\mu}x'_2.c'_2 \right\rangle & v'_{2i} &\triangleq \mu\alpha'_2. \left\langle \mathsf{K}_{2i}(\overrightarrow{\beta_{2i}}, \overrightarrow{y_{2i}}) \middle\| \tilde{\mu}x_2.c_2 \right\rangle \\ v_3 &\triangleq \mu\alpha_3. \left\langle \mathsf{K}'_3(\overrightarrow{\beta'_3}, \overrightarrow{y'_3}) \middle\| \tilde{\mu}x'_3.c'_3 \right\rangle & v'_3 &\triangleq \mu\alpha'_3. \left\langle \mathsf{K}_3(\overrightarrow{\beta_3}, \overrightarrow{y_3}) \middle\| \tilde{\mu}x_3.c_3 \right\rangle \end{aligned}$$

The composition of c and c' along α' and x' of type $\mathsf{F}'(\overrightarrow{C'})$ is equal to the identity command $\langle x \| \alpha \rangle$ via the combined strength of the $\tilde{\mu}$ and η axioms for the call-by-value data types F'_1, F'_2, and F'_3, as previously discussed in Section 8.1, as well as the call-by-value χ axiom to reassociate the bindings to bring the isomorphisms for those data types together, as follows:

$\langle \mu\alpha'.c \| \tilde{\mu}x'.c' \rangle$

$$=_{\eta_{\tilde{\mu}}}\left\langle\mu\alpha'.\left\langle x\,\middle\|\,\tilde{\mu}\left[\begin{array}{l}\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\left\langle v'_{1i}\,\middle\|\,\tilde{\mu}\left[\overrightarrow{\mathsf{K}'_{1i}(\overrightarrow{\beta'_{1j}},\overrightarrow{y'_{1j}}).\left\langle v'_3\,\middle\|\,\tilde{\mu}\left[\mathsf{K}'_3(\overrightarrow{\beta'_3},\overrightarrow{y'_3}).\left\langle\mathsf{K}'_{4j}(\overrightarrow{\beta'_{1j},\overrightarrow{\beta'_3},\overrightarrow{y'_3},\overrightarrow{y'_{1j}}})\,\middle\|\,\alpha'\right\rangle\right]\right\rangle}^{j}\right]\right\rangle}^{i}\\ \overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\left\langle v'_{2i}\,\middle\|\,\tilde{\mu}\left[\overrightarrow{\mathsf{K}'_{2i}(\overrightarrow{\beta'_{2j}},\overrightarrow{y'_{2j}}).\left\langle v'_3\,\middle\|\,\tilde{\mu}\left[\mathsf{K}'_3(\overrightarrow{\beta'_3},\overrightarrow{y'_3}).\left\langle\mathsf{K}'_{5j}(\overrightarrow{\beta'_{2j},\overrightarrow{\beta'_3},\overrightarrow{y'_3},\overrightarrow{y'_{2j}}})\,\middle\|\,\alpha'\right\rangle\right]\right\rangle}^{j}\right]\right\rangle}^{i}\end{array}\right]\right\rangle\right.$$

$$\left.\middle\|\,\tilde{\mu}\left[\begin{array}{l}\overrightarrow{\mathsf{K}'_{4i}(\overrightarrow{\beta'_{1i}},\overrightarrow{\beta'_3},\overrightarrow{y'_3},\overrightarrow{y'_{1i}}).\left\langle v_3\,\middle\|\,\tilde{\mu}\left[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\left\langle v_{1i}\,\middle\|\,\tilde{\mu}\left[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1j}},\overrightarrow{y_{1j}}).\left\langle\mathsf{K}_{4j}(\overrightarrow{\beta_{1j}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1j}})\,\middle\|\,\alpha\right\rangle}^{j}\right]\right\rangle\right]\right\rangle}^{i}\\ \overrightarrow{\mathsf{K}'_{5i}(\overrightarrow{\beta'_{2i}},\overrightarrow{\beta'_3},\overrightarrow{y'_3},\overrightarrow{y'_{2i}}).\left\langle v_3\,\middle\|\,\tilde{\mu}\left[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\left\langle v_{2i}\,\middle\|\,\tilde{\mu}\left[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2j}},\overrightarrow{y_{2j}}).\left\langle\mathsf{K}_{5j}(\overrightarrow{\beta_{1j}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1j}})\,\middle\|\,\alpha\right\rangle}^{j}\right]\right\rangle\right]\right\rangle}^{i}\end{array}\right]\right\rangle$$

$$=_{\mu_\alpha\tilde{\mu}_x\beta^{\mathsf{F}'}}\left\langle x\,\middle\|\,\tilde{\mu}\left[\begin{array}{l}\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\ \langle v'_{1i}|\ \tilde{\mu}\left[\overrightarrow{\mathsf{K}'_{1i}(\overrightarrow{\beta'_{1j}},\overrightarrow{y'_{1j}}).\langle v'_3|\ \tilde{\mu}\left[\mathsf{K}'_3(\overrightarrow{\beta'_3},\overrightarrow{y'_3}).\langle v_3|\ \tilde{\mu}[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\langle v_{1j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle\mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle]\rangle]}^{k}\Big\rangle\right]\Big\rangle}^{j}\right]\Big\rangle}^{i}\\ \overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\ \langle v'_{2i}|\ \tilde{\mu}\left[\overrightarrow{\mathsf{K}'_{2i}(\overrightarrow{\beta'_{2j}},\overrightarrow{y'_{2j}}).\langle v'_3|\ \tilde{\mu}\left[\mathsf{K}'_3(\overrightarrow{\beta'_3},\overrightarrow{y'_3}).\langle v_3|\ \tilde{\mu}[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\langle v_{2j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle\mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle]\rangle]}^{k}\Big\rangle\right]\Big\rangle}^{j}\right]\Big\rangle}^{i}\end{array}\right]\right\rangle$$

$$=_{\tilde{\mu}_{\mathcal{V}}\eta_{\mathcal{V}}^{\mathsf{F}'_3}}\left\langle x\,\middle\|\,\tilde{\mu}\left[\begin{array}{l}\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\langle\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3})\|\tilde{\mu}x_3.\ \langle v'_{1i}|\ \tilde{\mu}\left[\overrightarrow{\mathsf{K}'_{1i}(\overrightarrow{\beta'_{1j}},\overrightarrow{y'_{1j}}).\langle\mu\alpha'_3.c_3|\ \tilde{\mu}x'_3.\langle\mu\alpha_3.c'_3\|\tilde{\mu}[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\langle v_{1j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle\mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle]\rangle]\rangle}^{k}\Big\rangle}^{j}\right]\Big\rangle}^{i}\\ \overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\langle\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3})\|\tilde{\mu}x_3.\ \langle v'_{2i}|\ \tilde{\mu}\left[\overrightarrow{\mathsf{K}'_{2i}(\overrightarrow{\beta'_{2j}},\overrightarrow{y'_{2j}}).\langle\mu\alpha'_3.c_3|\ \tilde{\mu}x'_3.\langle\mu\alpha_3.c'_3\|\tilde{\mu}[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\langle v_{2j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle\mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle]\rangle]\rangle}^{k}\Big\rangle}^{j}\right]\Big\rangle}^{i}\end{array}\right]\right\rangle$$

$$=_{\chi\nu}\left\langle x \,\middle\|\, \tilde{\mu}\left[\begin{array}{l}
\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\langle\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3})\|\tilde{\mu}x_3.\,\langle v'_{1i}|}^{i}\\
\quad\left|\tilde{\mu}\left[\overrightarrow{\begin{array}{l}
\mathsf{K}'_{1i}(\overrightarrow{\beta'_{1j}},\overrightarrow{y'_{1j}}).\langle\mu\alpha_3.\,\langle\mu\alpha'_3.c_3\|\tilde{\mu}x'_3.c_3\rangle|\\
\quad\left|\tilde{\mu}[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\langle v_{1j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle\mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle}^{k}]\rangle]\right\rangle
\end{array}}^{j}\right]\right\rangle\\
\overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\langle\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3})\|\tilde{\mu}x_3.\,\langle v'_{2i}|}^{i}\\
\quad\left|\tilde{\mu}\left[\overrightarrow{\begin{array}{l}
\mathsf{K}'_{2i}(\overrightarrow{\beta'_{2j}},\overrightarrow{y'_{2j}}).\langle\mu\alpha_3.\,\langle\mu\alpha'_3.c_3\|\tilde{\mu}x'_3.c'_3\rangle|\\
\quad\left|\tilde{\mu}[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\langle v_{2j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle\mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle}^{k}]\rangle]\right\rangle
\end{array}}^{j}\right]\right\rangle
\end{array}\right]\right\rangle$$

$$=_{Iso\eta_{\mu}}\left\langle x \,\middle\|\, \tilde{\mu}\left[\begin{array}{l}
\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\langle\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3})\|\tilde{\mu}x_3.\,\langle v'_{1i}|}^{i}\\
\quad|\overrightarrow{\tilde{\mu}[\mathsf{K}'_{1i}(\overrightarrow{\beta'_{1j}},\overrightarrow{y'_{1j}}).\langle x_3\|\tilde{\mu}[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\langle v_{1j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle\mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle}^{k}]\rangle]\rangle]}^{j}\rangle\rangle\\
\overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\langle\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3})\|\tilde{\mu}x_3.\,\langle v'_{1i}|}^{i}\\
\quad|\overrightarrow{\tilde{\mu}[\mathsf{K}'_{2i}(\overrightarrow{\beta'_{2j}},\overrightarrow{y'_{2j}}).\langle x_3\|\tilde{\mu}[\mathsf{K}_3(\overrightarrow{\beta_3},\overrightarrow{y_3}).\langle v_{2j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle\mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle}^{k}]\rangle]\rangle]}^{j}\rangle\rangle
\end{array}\right]\right\rangle$$

$$=_{\tilde{\mu}_x\mu_\alpha\beta^{\mathsf{F}_3}}\left\langle x \,\middle\|\, \tilde{\mu}\left[\begin{array}{l}
\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\,\langle v'_{1i}|}^{i}\\
\quad\left|\overrightarrow{\tilde{\mu}[\mathsf{K}'_{1i}(\overrightarrow{\beta'_{1j}},\overrightarrow{y'_{1j}}).\langle v_{1j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle\mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle]}^{k}\rangle\,]}^{j}\right\rangle\\
\overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\,\langle v'_{2i}|}^{i}\\
\quad\left|\overrightarrow{\tilde{\mu}[\mathsf{K}'_{2i}(\overrightarrow{\beta'_{2j}},\overrightarrow{y'_{2j}}).\langle v_{2j}\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle\mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle]}^{k}\rangle\,]}^{j}\right\rangle
\end{array}\right]\right\rangle$$

$$=_{\tilde{\mu}\nu\eta_{\nu}^{\mathsf{F}'_1}\eta_{\nu}^{\mathsf{F}'_2}}\left\langle x \,\middle\|\, \tilde{\mu}\left[\begin{array}{l}
\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\,\langle\mathsf{K}_{1i}(\overrightarrow{\beta_{1i}},\overrightarrow{y_{1i}})|}^{i}\\
\quad\left|\tilde{\mu}x_1.\langle\mu\alpha'_1.c_1\|\tilde{\mu}x'_1.\langle\mu\alpha_1.c'_1\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle\mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle]}^{k}\rangle\rangle\right\rangle\\
\overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\,\langle\mathsf{K}_{2i}(\overrightarrow{\beta_{2i}},\overrightarrow{y_{2i}})|}^{i}\\
\quad\left|\tilde{\mu}x_2.\langle\mu\alpha'_2.c_2\|\tilde{\mu}x'_2.\langle\mu\alpha_2.c'_2\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle\mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle]}^{k}\rangle\rangle\right\rangle
\end{array}\right]\right\rangle$$

$$=_{\chi\nu}\left\langle x \,\middle\|\, \tilde{\mu}\left[\begin{array}{l}
\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\,\langle\mathsf{K}_{1i}(\overrightarrow{\beta_{1i}},\overrightarrow{y_{1i}})|}^{i}\\
\quad\left|\tilde{\mu}x_1.\langle\mu\alpha_1.\,\langle\mu\alpha'_1.c_1\|\tilde{\mu}x'_1.c'_1\rangle\,\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle\mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle]}^{k}\rangle\right\rangle\\
\overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\,\langle\mathsf{K}_{2i}(\overrightarrow{\beta_{2i}},\overrightarrow{y_{2i}})|}^{i}\\
\quad\left|\tilde{\mu}x_2.\langle\mu\alpha_2.\,\langle\mu\alpha'_2.c_2\|\tilde{\mu}x'_2.c'_2\rangle\,\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle\mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle]}^{k}\rangle\right\rangle
\end{array}\right]\right\rangle$$

$$=_{Iso}\left\langle x \middle\| \tilde{\mu}\begin{bmatrix}\overrightarrow{\begin{array}{l}\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\,\langle \mathsf{K}_{1i}(\overrightarrow{\beta_{1i}},\overrightarrow{y_{1i}})| \\ \quad \left|\tilde{\mu}x_1.\langle \mu\alpha_1.\,\langle x_1\|\alpha_1\rangle\,\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle \mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle}^{k}]\rangle\right\rangle\end{array}}^{i} \\ \overrightarrow{\begin{array}{l}\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\,\langle \mathsf{K}_{2i}(\overrightarrow{\beta_{2i}},\overrightarrow{y_{2i}})| \\ \quad \left|\tilde{\mu}x_2.\langle \mu\alpha_2.\,\langle x_2\|\alpha_2\rangle\,\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle \mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle}^{k}]\rangle\right\rangle\end{array}}^{i}\end{bmatrix}\right\rangle$$

$$=_{\eta_\mu\eta_{\tilde{\mu}}}\left\langle x \middle\| \tilde{\mu}\begin{bmatrix}\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\langle \mathsf{K}_{1i}(\overrightarrow{\beta_{1i}},\overrightarrow{y_{1i}})\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{1j}(\overrightarrow{\beta_{1k}},\overrightarrow{y_{1k}}).\langle \mathsf{K}_{4k}(\overrightarrow{\beta_{1k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1k}})\|\alpha\rangle}^{k}]\rangle}^{i} \\ \overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\langle \mathsf{K}_{2i}(\overrightarrow{\beta_{2i}},\overrightarrow{y_{2i}})\|\tilde{\mu}[\overrightarrow{\mathsf{K}_{2j}(\overrightarrow{\beta_{2k}},\overrightarrow{y_{2k}}).\langle \mathsf{K}_{5k}(\overrightarrow{\beta_{2k}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2k}})\|\alpha\rangle}^{k}]\rangle}^{i}\end{bmatrix}\right\rangle$$

$$=_{\mu\tilde{\mu}\beta^{\mathsf{F}_1}\beta^{\mathsf{F}_2}}\left\langle x \middle\| \tilde{\mu}\begin{bmatrix}\overrightarrow{\mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{1i}}).\langle \mathsf{K}_{4i}(\overrightarrow{\beta_{1i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{1i}})\|\alpha\rangle}^{i} \\ \overrightarrow{\mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{y_{2i}}).\langle \mathsf{K}_{5i}(\overrightarrow{\beta_{2i}},\overrightarrow{\beta_3},\overrightarrow{y_3},\overrightarrow{\beta_{2i}})\|\alpha\rangle}^{i}\end{bmatrix}\right\rangle$$

$$=_{\eta^{\mathsf{F}}}\langle x\|\alpha\rangle$$

And the reverse composition of c' and c along α and x of type $\mathsf{F}(\overrightarrow{C})$ is equal to the identity command $\langle x'\|\alpha'\rangle$ similarly. □

Lemma 8.4 ((Co-)Data interchange shift instance)**.** *For any types $\overrightarrow{C:\mathcal{S}}$, $\overrightarrow{C':\mathcal{S}'}$ and (co-)data declarations*

$$\begin{array}{ll}\mathbf{data}\,\mathsf{F}(\overrightarrow{X:\mathcal{S}}):\mathcal{T}\,\mathbf{where} & \mathbf{data}\,\mathsf{F}'(\overrightarrow{X':\mathcal{S}'}):\mathcal{T}\,\mathbf{where} \\ \quad\overrightarrow{\mathsf{K}:\left(\Gamma\vdash\mathsf{F}(\overrightarrow{X:\mathcal{S}})\mid\Delta\right)} & \quad\overrightarrow{\mathsf{K}':\left(\Gamma'\vdash\mathsf{F}'(\overrightarrow{X':\mathcal{S}'})\mid\Delta'\right)} \\ \mathbf{codata}\,\mathsf{G}(\overrightarrow{X:\mathcal{S}}):\mathcal{R}\,\mathbf{where} & \mathbf{codata}\,\mathsf{G}'(\overrightarrow{X':\mathcal{S}'}):\mathcal{R}\,\mathbf{where} \\ \quad\overrightarrow{\mathsf{O}:\left(\Gamma\mid\mathsf{F}(\overrightarrow{X:\mathcal{S}})\vdash\Delta\right)} & \quad\overrightarrow{\mathsf{O}':\left(\Gamma'\mid\mathsf{G}'(\overrightarrow{X':\mathcal{S}'})\vdash\Delta'\right)}\end{array}$$

$\mathsf{F}(\overrightarrow{C})\approx\mathsf{F}'(\overrightarrow{C'})$ *implies* $\mathsf{G}(\overrightarrow{C})\approx\mathsf{G}'(\overrightarrow{C'})$ *when* $\mathcal{T}=\mathcal{V}$ *and* $\mathsf{F}(\overrightarrow{C})\approx\mathsf{F}'(\overrightarrow{C'})$ *implies* $\mathsf{G}(\overrightarrow{C})\approx\mathsf{G}'(\overrightarrow{C'})$ *when* $\mathcal{R}=\mathcal{N}$*.*

Proof. First, suppose that the commands

$$c_1:(x_1:\mathsf{F}(\overrightarrow{C})\vdash\alpha'_1:\mathsf{F}'(\overrightarrow{C'}))\qquad\text{and}\qquad c'_1:(x'_1:\mathsf{F}'(\overrightarrow{C'})\vdash\alpha_1:\mathsf{F}(\overrightarrow{C}))$$

witness the isomorphism $\mathsf{F}(\overrightarrow{C}) \approx \mathsf{F}'(\overrightarrow{C'})$. Then the isomorphism between $\mathsf{G}(\overrightarrow{C})$ and $\mathsf{G}'(\overrightarrow{C'})$ is established by:

$$\begin{aligned}
c_2 &\triangleq \left\langle \mu\left(\overrightarrow{\mathsf{O}'_i[\overrightarrow{y'_i},\overrightarrow{\beta'_i}].\left\langle \mu\alpha_1.\left\langle \mathsf{K}'_i(\overrightarrow{\beta'_i},\overrightarrow{y'_i}) \middle\| \tilde{\mu}x'_1.c'_1\right\rangle \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_j(\overrightarrow{\beta_j},\overrightarrow{y_j}).\langle x_2 \| \mathsf{O}_j[\overrightarrow{y_j},\overrightarrow{\beta_j}]\rangle}^{j}\right]\right\rangle}^{i}\right) \middle\| \alpha'_2 \right\rangle \\
&\quad : (x_2 \colon \mathsf{G}(\overrightarrow{C}) \vdash \alpha'_2 \colon \mathsf{G}'(\overrightarrow{C'})) \\
c'_2 &\triangleq \left\langle \mu\left(\overrightarrow{\mathsf{O}_i[\overrightarrow{y_i},\overrightarrow{\beta_i}].\left\langle \mu\alpha'_1.\left\langle \mathsf{K}_i(\overrightarrow{\beta_i},\overrightarrow{y_i}) \middle\| \tilde{\mu}x_1.c_1\right\rangle \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}'_j(\overrightarrow{\beta'_j},\overrightarrow{y'_j}).\langle x'_2 \| \mathsf{O}'_j[\overrightarrow{y'_j},\overrightarrow{\beta'_j}]\rangle}^{j}\right]\right\rangle}^{i}\right) \middle\| \alpha_2 \right\rangle \\
&\quad : (x'_2 \colon \mathsf{G}'(\overrightarrow{C'}) \vdash \alpha_2 \colon \mathsf{G}(\overrightarrow{C}))
\end{aligned}$$

The composition of c'_2 and c_2 along α_2 and x_2 of type $\mathsf{G}(\overrightarrow{C})$ is equal to the identity command $\langle x'_2 \| \alpha'_2 \rangle$ via the β^{G}, $\eta^{\mathsf{F}}_{\mathcal{V}}$, $\beta^{\mathsf{F}'}$, and $\eta^{\mathsf{G}'}$ axioms as follows:

$$\begin{aligned}
&\langle \mu\alpha_2.c'_2 \| \tilde{\mu}x_2.c_2 \rangle \\
&\triangleq \left\langle \mu\alpha_2.\left\langle \mu\left(\overrightarrow{\mathsf{O}_i[\overrightarrow{y_i},\overrightarrow{\beta_i}].\left\langle \mu\alpha'_1.\left\langle \mathsf{K}_i(\overrightarrow{\beta_i},\overrightarrow{y_i}) \middle\| \tilde{\mu}x_1.c_1\right\rangle \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}'_j(\overrightarrow{\beta'_j},\overrightarrow{y'_j}).\langle x'_2 \| \mathsf{O}'_j[\overrightarrow{y'_j},\overrightarrow{\beta'_j}]\rangle}^{j}\right]\right\rangle}^{i}\right) \middle\| \alpha_2 \right\rangle \right. \\
&\qquad \left. \middle\| \tilde{\mu}x_2.\left\langle \mu\left(\overrightarrow{\mathsf{O}'_i[\overrightarrow{y'_i},\overrightarrow{\beta'_i}].\left\langle \mu\alpha_1.\left\langle \mathsf{K}'_i(\overrightarrow{\beta'_i},\overrightarrow{y'_i}) \middle\| \tilde{\mu}x'_1.c'_1\right\rangle \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_j(\overrightarrow{\beta_j},\overrightarrow{y_j}).\langle x_2 \| \mathsf{O}_j[\overrightarrow{y_j},\overrightarrow{\beta_j}]\rangle}^{j}\right]\right\rangle}^{i}\right) \middle\| \alpha'_2 \right\rangle \right\rangle \\
&=_{\eta_\mu} \left\langle \mu\left(\overrightarrow{\mathsf{O}_i[\overrightarrow{y_i},\overrightarrow{\beta_i}].\left\langle \mu\alpha'_1.\left\langle \mathsf{K}_i(\overrightarrow{\beta_i},\overrightarrow{y_i}) \middle\| \tilde{\mu}x_1.c_1\right\rangle \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}'_j(\overrightarrow{\beta'_j},\overrightarrow{y'_j}).\langle x'_2 \| \mathsf{O}'_j[\overrightarrow{y'_j},\overrightarrow{\beta'_j}]\rangle}^{j}\right]\right\rangle}^{i}\right) \right. \\
&\qquad \left. \middle\| \tilde{\mu}x_2.\left\langle \mu\left(\overrightarrow{\mathsf{O}'_i[\overrightarrow{y'_i},\overrightarrow{\beta'_i}].\left\langle \mu\alpha_1.\left\langle \mathsf{K}'_i(\overrightarrow{\beta'_i},\overrightarrow{y'_i}) \middle\| \tilde{\mu}x'_1.c'_1\right\rangle \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_j(\overrightarrow{\beta_j},\overrightarrow{y_j}).\langle x_2 \| \mathsf{O}_j[\overrightarrow{y_j},\overrightarrow{\beta_j}]\rangle}^{j}\right]\right\rangle}^{i}\right) \middle\| \alpha'_2 \right\rangle \right\rangle \\
&=_{\tilde{\mu}_{\mathcal{N}}} \left\langle \mu\left(\overrightarrow{\begin{array}{l}
\mathsf{O}'_i[\overrightarrow{y'_i},\overrightarrow{\beta'_i}].\ \left\langle \mu\alpha_1.\left\langle \mathsf{K}'_i(\overrightarrow{\beta'_i},\overrightarrow{y'_i}) \middle\| \tilde{\mu}x'_1.c'_1\right\rangle \right. \\
\qquad \left. \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_j(\overrightarrow{\beta_j},\overrightarrow{y_j}).\left\langle \mu\left(\overrightarrow{\begin{array}{l}
\mathsf{O}_k[\overrightarrow{y_k},\overrightarrow{\beta_k}].\ \left\langle \mu\alpha'_1.\left\langle \mathsf{K}_k(\overrightarrow{\beta_k},\overrightarrow{y_k}) \middle\| \tilde{\mu}x_1.c_1\right\rangle \right. \\
\qquad \left. \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}'_l(\overrightarrow{\beta'_l},\overrightarrow{y'_l}).\langle x'_2 \| \mathsf{O}'_l[\overrightarrow{y'_l},\overrightarrow{\beta'_l}]\rangle}^{l}\right]\right\rangle
\end{array}}^{k}\right) \middle\| \mathsf{O}_j[\overrightarrow{y_j},\overrightarrow{\beta_j}]\right\rangle}^{j}\right]\right\rangle
\end{array}}^{i}\right) \middle\| \alpha'_2 \right\rangle
\end{aligned}$$

$$=_{\beta^{\mathsf{G}}\mu_\alpha\tilde\mu_x}\Big\langle\mu\Big(\overrightarrow{\begin{array}{l}\mathsf{O}'_i[\vec{y'_i},\vec{\beta'_i}].\ \langle\mu\alpha_1.\langle\mathsf{K}'_i(\vec{\beta'_i},\vec{y'_i})\|\tilde\mu x'_1.c'_1\rangle\\ \qquad\|\tilde\mu\Big[\overrightarrow{\begin{array}{l}\mathsf{K}_j(\vec{\beta_j},\vec{y_j}).\ \langle\mu\alpha'_1.\ \langle\mathsf{K}_j(\vec{\beta_j},\vec{y_j})\|\tilde\mu x_1.c_1\rangle\\ \qquad\|\tilde\mu\Big[\overrightarrow{\mathsf{K}'_l(\vec{\beta'_l},\vec{y'_l}).\langle x'_2\|\,\mathsf{O}'_l[\vec{y'_l},\vec{\beta'_l}]\rangle}^{l}\Big]\rangle\end{array}}^{j}\Big]\rangle\end{array}}^{i}\Big)\Big\|\alpha'_2\Big\rangle$$

$$=_{\tilde\mu_{\mathcal{V}}\eta^{\mathsf{F}}_{\mathcal{V}}}\Big\langle\mu\Big(\overrightarrow{\begin{array}{l}\mathsf{O}'_i[\vec{y'_i},\vec{\beta'_i}].\ \langle\mu\alpha_1.\langle\mathsf{K}'_i(\vec{\beta'_i},\vec{y'_i})\|\tilde\mu x'_1.c'_1\rangle\\ \qquad\|\tilde\mu x_1.\ \langle\mu\alpha'_1.c_1\|\tilde\mu\Big[\overrightarrow{\mathsf{K}'_l(\vec{\beta'_l},\vec{y'_l}).\langle x'_2\|\,\mathsf{O}'_l[\vec{y'_l},\vec{\beta'_l}]\rangle}^{l}\Big]\rangle\rangle\end{array}}^{i}\Big)\Big\|\alpha'_2\Big\rangle$$

$$=_{\chi_{\mathcal{V}}}\Big\langle\mu\Big(\overrightarrow{\mathsf{O}'_i[\vec{y'_i},\vec{\beta'_i}].\Big\langle\mathsf{K}'_i(\vec{\beta'_i},\vec{y'_i})\Big\|\begin{array}{l}\tilde\mu x'_1.\ \langle\mu\alpha'_1.\ \langle\mu\alpha_1.c'_1\|\tilde\mu x_1.c_1\rangle\\ \qquad\|\tilde\mu\Big[\overrightarrow{\mathsf{K}'_l(\vec{\beta'_l},\vec{y'_l}).\langle x'_2\|\,\mathsf{O}'_l[\vec{y'_l},\vec{\beta'_l}]\rangle}^{l}\Big]\rangle\end{array}\Big\rangle}^{i}\Big)\Big\|\alpha'_2\Big\rangle$$

$$=_{Iso}\Big\langle\mu\Big(\overrightarrow{\mathsf{O}'_i[\vec{y'_i},\vec{\beta'_i}].\Big\langle\mathsf{K}'_i(\vec{\beta'_i},\vec{y'_i})\Big\|\tilde\mu x'_1.\Big\langle\mu\alpha'_1.\ \langle x'_1\|\alpha'_1\rangle\Big\|\tilde\mu\Big[\overrightarrow{\mathsf{K}'_l(\vec{\beta'_l},\vec{y'_l}).\langle x'_2\|\,\mathsf{O}'_l[\vec{y'_l},\vec{\beta'_l}]\rangle}^{l}\Big]\Big\rangle\Big\rangle}^{i}\Big)\Big\|\alpha'_2\Big\rangle$$

$$=_{\eta_\mu\eta_{\tilde\mu}}\Big\langle\mu\Big(\overrightarrow{\mathsf{O}'_i[\vec{y'_i},\vec{\beta'_i}].\Big\langle\mathsf{K}'_i(\vec{\beta'_i},\vec{y'_i})\Big\|\tilde\mu\Big[\overrightarrow{\mathsf{K}'_l(\vec{\beta'_l},\vec{y'_l}).\langle x'_2\|\,\mathsf{O}'_l[\vec{y'_l},\vec{\beta'_l}]\rangle}^{l}\Big]\Big\rangle}^{i}\Big)\Big\|\alpha'_2\Big\rangle$$

$$=_{\beta^{\mathsf{F}'}\mu\tilde\mu}\Big\langle\mu\Big(\overrightarrow{\mathsf{O}'_i[\vec{y'_i},\vec{\beta'_i}].\langle x'_2\|\,\mathsf{O}'_i[\vec{y'_i},\vec{\beta'_i}]\rangle}^{i}\Big)\Big\|\alpha'_2\Big\rangle$$

$$=_{\eta^{\mathsf{G}'}}\langle x'_2\|\alpha'_2\rangle$$

The composition of c_2 and c'_2 along α'_2 and x'_2 of type $\mathsf{G}'(\vec{C})$ is equal to the identity command $\langle x_2\|\alpha_2\rangle$ via the $\beta^{\mathsf{G}'}$, $\eta^{\mathsf{F}'}_{\mathcal{V}}$, β^{F}, and η^{G} similarly.

Second, suppose that the commands $c_2 : (x_2 : \mathsf{G}(\vec{C}) \vdash \alpha'_2 : \mathsf{G}'(\vec{C'}))$ and $c'_2 : (x'_2 : \mathsf{G}'(\vec{C'}) \vdash \alpha_2 : \mathsf{G}(\vec{C}))$ witness the isomorphism $\mathsf{G}(\vec{C}) \approx \mathsf{G}'(\vec{C'})$. Then the isomorphism between $\mathsf{F}(\vec{C})$ and $\mathsf{F}'(\vec{C'})$ is established by:

$$c_1 \triangleq \Big\langle x_1\Big\|\tilde\mu\Big[\overrightarrow{\mathsf{K}_i(\vec{\beta_i},\vec{y_i}).\Big\langle\mu\Big(\overrightarrow{\mathsf{O}'_j[\vec{y'_j}\vec{\beta'_j}].\langle\mathsf{K}'_j(\vec{y'_j},\vec{\beta'_j})\|\alpha'_1\rangle}^{j}\Big)\Big\|\tilde\mu x'_2.\ \langle\tilde\mu\alpha_2.c'_2\|\mathsf{O}_i[\vec{y_i},\vec{\beta_i}]\rangle\Big\rangle}^{i}\Big]\Big\rangle$$

$$: (x_1 \!:\mathsf{F}(\vec{C}) \vdash \alpha'_1 \!:\mathsf{F}'(\vec{C'}))$$

$$c'_1 \triangleq \Big\langle x'_1\Big\|\tilde\mu\Big[\overrightarrow{\mathsf{K}_i(\vec{\beta_i},\vec{y_i}).\Big\langle\mu\Big(\overrightarrow{\mathsf{O}_j[\vec{y_j}\vec{\beta_j}].\langle\mathsf{K}_j(\vec{y_j},\vec{\beta_j})\|\alpha_1\rangle}^{j}\Big)\Big\|\tilde\mu x_2.\ \langle\tilde\mu\alpha'_2.c_2\|\mathsf{O}'_i[\vec{y'_i},\vec{\beta'_i}]\rangle\Big\rangle}^{i}\Big]\Big\rangle$$

$$: (x'_1 \!:\mathsf{F}'(\vec{C'}) \vdash \alpha_1 \!:\mathsf{F}(\vec{C}))$$

Both compositions of c and c' are equal to the identity command analogously to the previous part by duality. □

With the above two isomorphisms established, we now have enough to justify the soundness of all the proposed structural laws for (co-)data declarations.

Theorem 8.6 (Structural law soundness)**.** *The declaration isomorphism laws in Figures 8.4 and 8.5 are all sound.*

Proof. The data laws all follow by generalizing the particular instances where the data types are isomorphic: the commute, interchange, and compatibility laws are all immediate consequence of Lemmas 8.1, 8.4 and 8.3, both mix laws follow from Lemma 8.2 by taking either F_1 and F'_1 to be the empty data declaration of no alternatives or taking F_3 and F'_3 to be the unit data declaration of one alternative with no components (both of which are isomorphic by reflexivity), and the shift law follows by applying Lemma 8.4 twice. The co-data laws follow from the data laws by Lemma 8.4. □

Finally, there is one more property about (co-)data declarations that will be extremely useful in the following section. Namely that certain singleton (co-)data types are just trivial wrappers around another type. In the right circumstances, these wrappers can be identified with their underlying types, up to isomorphism, which lets us connect the world of (co-)data declarations with the world of actual types.

Lemma 8.5 ((Co-)Data identity)**.**

a) For any $\mathbf{data}\,\mathsf{F}(\Theta) : \mathcal{R}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{T} \vdash \mathsf{F}(\Theta) \mid)$*, if either* $\mathcal{T} = \mathcal{V}$ *or* $\mathcal{R} = \mathcal{V}$ *then* $\vec{\Theta} \vDash \mathsf{F}(\Theta) \approx A$*.*

b) For any $\mathbf{codata}\,\mathsf{G}(\Theta) : \mathcal{R}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}(\Theta) \vdash A : \mathcal{T})$*, if either* $\mathcal{T} = \mathcal{N}$ *or* $\mathcal{R} = \mathcal{N}$ *then* $\vec{\Theta} \vDash \mathsf{G}(\vec{\Theta}) \approx A$*.*

Proof. a) Let $\Theta = \overrightarrow{X : \mathcal{S}}$, suppose $\overrightarrow{C : \mathcal{S}}$, and let $B = A\{\overrightarrow{C/X}\}$. $\mathsf{F}(\vec{C}) \approx B$ is established by the commands:

$$c_1 \triangleq \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle y \| \beta \rangle] \rangle : (x : \mathsf{F}(\vec{C}) \vdash \beta : B) \quad c_2 \triangleq \langle \mathsf{K}(y) \| \alpha \rangle : (y : B \vdash \alpha : \mathsf{F}(\vec{C}))$$

First, the composition of c_2 and c_1 along α and x of type $\mathsf{F}(\vec{C}) : \mathcal{R}$ is equal to the identity command $\langle y \| \beta \rangle$ by using the η_μ and $\eta_{\tilde{\mu}}$ axioms to reveal the β^{F} redex as follows:

$$\begin{aligned}
\langle \mu\alpha{::}\mathcal{R}.c_2 \| \tilde{\mu}x{::}\mathcal{R}.c_1 \rangle &\triangleq \langle \mu\alpha{::}\mathcal{R}.\, \langle \mathsf{K}(y) \| \alpha \rangle \| \tilde{\mu}x{::}\mathcal{R}.\, \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle y \| \beta \rangle] \rangle \rangle \\
&=_{\eta_\mu \eta_{\tilde{\mu}}} \langle \mathsf{K}(y) \| \tilde{\mu}[\mathsf{K}(y).\langle y \| \beta \rangle] \rangle \\
&=_{\beta^{\mathsf{F}}} \langle y \| \tilde{\mu}y{::}\mathcal{T}.\, \langle y \| \beta \rangle \rangle =_{\tilde{\mu}_{\mathcal{T}}} \langle y \| \beta \rangle
\end{aligned}$$

Next, suppose that $\mathcal{T} = \mathcal{V}$. The composition of c_1 and c_2 along β and y of type $B : \mathcal{V}$ is equal to the identity command $\langle x \| \alpha \rangle$ by using the strength of the $\mu_{\mathcal{V}}$ axiom to reveal the η^{F} redex as follows:

$$\begin{aligned}
\langle \mu\beta{::}\mathcal{V}.c_1 \| \tilde{\mu}y{::}\mathcal{V}.c_2 \rangle &\triangleq \langle \mu\beta{::}\mathcal{V}.\, \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle y \| \beta \rangle] \rangle \| \tilde{\mu}y{::}\mathcal{V}.\, \langle \mathsf{K}(y) \| \alpha \rangle \rangle \\
&=_{\mu_{\mathcal{V}}} \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle y{::}\mathcal{V} \| \tilde{\mu}y.\, \langle \mathsf{K}(y) \| \alpha \rangle \rangle] \rangle \\
&=_{\tilde{\mu}_{\mathcal{V}}} \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle \mathsf{K}(y) \| \alpha \rangle] \rangle =_{\eta^{\mathsf{F}}} \langle x \| \alpha \rangle
\end{aligned}$$

Otherwise, suppose that $\mathcal{R} = \mathcal{V}$. The composition of c_1 and c_2 along β and y of type $B : \mathcal{T}$ is equal to the identity command $\langle x \| \alpha \rangle$ by using the combined strength of the $\mu_{\mathcal{V}}$ and η^{F} axioms to percolate out the case analysis on x and create an inner β^{F} redex:

$$\begin{aligned}
&\langle \mu\beta{::}\mathcal{T}.c_1 \| \tilde{\mu}y{::}\mathcal{T}.c_2 \rangle \\
&\triangleq \langle \mu\beta{::}\mathcal{T}.\, \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle y \| \beta \rangle] \rangle \| \tilde{\mu}y{::}\mathcal{T}.\, \langle \mathsf{K}(y) \| \alpha \rangle \rangle \\
&=_{\mu_{\mathcal{V}}} \langle x \| \tilde{\mu}x.\, \langle \mu\beta{::}\mathcal{T}.\, \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle y \| \beta \rangle] \rangle \| \tilde{\mu}y{::}\mathcal{T}.\, \langle \mathsf{K}(y) \| \alpha \rangle \rangle \rangle \\
&=_{\mu_{\mathcal{V}}\eta^{\mathsf{F}}} \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle \mathsf{K}(y) \| \tilde{\mu}x.\, \langle \mu\beta{::}\mathcal{T}.\, \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle y \| \beta \rangle] \rangle \| \tilde{\mu}y{::}\mathcal{T}.\, \langle \mathsf{K}(y) \| \alpha \rangle \rangle \rangle] \rangle \\
&=_{\tilde{\mu}_{\mathcal{V}}} \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle \mu\beta{::}\mathcal{T}.\, \langle \mathsf{K}(y) \| \tilde{\mu}[\mathsf{K}(y).\langle y \| \beta \rangle] \rangle \| \tilde{\mu}y{::}\mathcal{T}.\, \langle \mathsf{K}(y) \| \alpha \rangle \rangle] \rangle \\
&=_{\beta^{\mathsf{F}}\tilde{\mu}_{\mathcal{T}}} \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle \mu\beta{::}\mathcal{T}.\, \langle y \| \beta \rangle \| \tilde{\mu}y{::}\mathcal{T}.\, \langle \mathsf{K}(y) \| \alpha \rangle \rangle] \rangle \\
&=_{\eta_\mu \tilde{\mu}_{\mathcal{T}}} \langle x \| \tilde{\mu}[\mathsf{K}(y).\langle \mathsf{K}(y) \| \alpha \rangle] \rangle =_{\eta^{\mathsf{F}}} \langle x \| \alpha \rangle
\end{aligned}$$

b) Analogous to the proof of Lemma 8.5 (a) by duality. □

Internal polarized laws of declarations

Now that we have established some basic structural laws about isomorphisms between general user-defined (co-)data types, we can focus on some more specific laws about the polarized types in Figure 8.1. In particular, we can show that these polar types play a part in a family of isomorphisms that closely resemble some of the logical rules of the sequent calculus. Namely, each of the left rules for the positive data types and the right rules for the negative co-data types correspond to an isomorphism between (co-)data declarations with signatures matching the premises and conclusion of the rules, as shown in Figures 8.6 and 8.7. The role of using declarations for this purpose is to give enough structural substrate for stating these rules: the sequents containing multiple inputs and multiple outputs in the rules can be expressed by the types of constructors or observers, and multiple premises can be expressed by multiple alternatives for constructors or observers. And as a result, we can reason about the polarized types as sub-components within the structure of larger (co-)data types.

Theorem 8.7 (Polarized sub-structure laws)**.** *The declaration isomorphism laws in Figures 8.6 and 8.7 are all sound.*

Proof. Due to the (co-)data interchange laws from Figures 8.4 and 8.5 we only need to demonstrate half of the isomorphisms in Figures 8.6 and 8.7 since each side implies the other. So let us focus only on the more familiar data type declarations, because all the laws for polarized co-data sub-structures are derived from those. In each case, the main technique for establishing these laws is that, for any substitution θ matching the environment Θ, the data type $\mathsf{F}'(\Theta)\theta$ on each right-hand side is isomorphic to the single component of the single alternative under the substitution θ according to Lemma 8.5 because each of the data types are call-by-value (i.e. $\mathsf{F}'(\Theta) : \mathcal{V}$). What remains is to then demonstrate that in each case, the data type $\mathsf{F}(\Theta)\theta$ is *also* isomorphic to that same type.

The sub-structure laws for the nullary data types $(0, 1)$ are the easiest to show. Note how for the $0L$ law we directly have that $\mathsf{F}(\Theta) \approx 0$ as a trivial case of Lemma 8.1 (b), and $\mathsf{F}'(\Theta) \approx 0$ by Lemma 8.5 (a), so together we know $\mathsf{F}(\Theta) \approx 0 \approx \mathsf{F}'(\Theta)$. Similarly for the $1L$ law, we have $\mathsf{F}(\Theta) \approx 1$ as a trivial case of Lemma 8.1 (a), and so we get $\mathsf{F}(\Theta) \approx 1 \approx \mathsf{F}'(\Theta)$ from Lemma 8.5 (a) as well.

The sub-structural laws for the unary data types $(\sim, \downarrow_{\mathcal{S}})$ follow a different line of reasoning, but are not much more difficult to demonstrate. For instance, consider

Additive laws

$$\begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K}_1 : (A : \mathcal{V} \vdash \mathsf{F}(\Theta) \mid) \\ \qquad \mathsf{K}_2 : (B : \mathcal{V} \vdash \mathsf{F}(\Theta) \mid) \end{array} \approx_{\oplus L} \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K}' : (A \oplus B : \mathcal{V} \vdash \mathsf{F}'(\Theta) \mid) \end{array}$$

$$\textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \approx_{0L} \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K}' : (0 : \mathcal{V} \vdash \mathsf{F}'(\Theta) \mid) \end{array}$$

Multiplicative laws

$$\begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K} : (A : \mathcal{V}, B : \mathcal{V} \vdash \mathsf{F}(\Theta) \mid) \end{array} \approx_{\otimes L} \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K}' : (A \otimes B : \mathcal{V} \vdash \mathsf{F}'(\Theta) \mid) \end{array}$$

$$\begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K} : (\vdash \mathsf{F}(\Theta) \mid) \end{array} \approx_{1L} \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K}' : (1 : \mathcal{V} \vdash \mathsf{F}'(\Theta) \mid) \end{array}$$

Negation Laws

$$\begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K} : (\vdash \mathsf{F}(\Theta) \mid A : \mathcal{N}) \end{array} \approx_{\sim L} \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K}' : ({\sim}A : \mathcal{V} \vdash \mathsf{F}(\Theta) \mid) \end{array}$$

Shift Laws

$$\begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K} : (A : \mathcal{S} \vdash \mathsf{F}(\Theta) \mid) \end{array} \approx_{\downarrow_{\mathcal{S}} L} \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \qquad \mathsf{K}' : ({\downarrow_{\mathcal{S}}} A : \mathcal{V} \vdash \mathsf{F}'(\Theta) \mid) \end{array}$$

FIGURE 8.6. Isomorphism laws of positively polarized data sub-structures.

Additive laws

$$\begin{array}{rcl} \begin{array}{l}\textbf{codata}\,\mathsf{G}(\Theta) : \mathcal{V}\,\textbf{where}\\ \quad \mathsf{O}_1 : (\,|\,\mathsf{G}(\Theta) \vdash A : \mathcal{N})\\ \quad \mathsf{O}_2 : (\,|\,\mathsf{G}(\Theta) \vdash B : \mathcal{N})\end{array} & \approx_{\& R} & \begin{array}{l}\textbf{codata}\,\mathsf{G}'(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O}' : (\,|\,\mathsf{G}'(\Theta) \vdash A \mathbin{\&} B : \mathcal{N})\end{array} \end{array}$$

$$\begin{array}{rcl} \textbf{codata}\,\mathsf{G}(\Theta) : \mathcal{N}\,\textbf{where} & \approx_{\top R} & \begin{array}{l}\textbf{codata}\,\mathsf{G}'(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O}' : (\,|\,\mathsf{G}'(\Theta) \vdash \top : \mathcal{N})\end{array} \end{array}$$

Multiplicative laws

$$\begin{array}{rcl} \begin{array}{l}\textbf{codata}\,\mathsf{G}(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O} : (\vdash \mathsf{G}(\Theta) \mid A : \mathcal{N}, B : \mathcal{N})\end{array} & \approx_{\parr R} & \begin{array}{l}\textbf{codata}\,\mathsf{G}'(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O}' : (\vdash \mathsf{G}'(\Theta) \mid A \parr B : \mathcal{N})\end{array} \end{array}$$

$$\begin{array}{rcl} \begin{array}{l}\textbf{codata}\,\mathsf{G}(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O} : (\,|\,\mathsf{G}(\Theta) \vdash\,)\end{array} & \approx_{\bot R} & \begin{array}{l}\textbf{codata}\,\mathsf{G}'(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O}' : (\,|\,\mathsf{G}'(\Theta) \vdash \bot : \mathcal{N})\end{array} \end{array}$$

Negation Laws

$$\begin{array}{rcl} \begin{array}{l}\textbf{codata}\,\mathsf{G}(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O} : (A : \mathcal{V} \mid \mathsf{G}(\Theta) \vdash\,)\end{array} & \approx_{\neg R} & \begin{array}{l}\textbf{codata}\,\mathsf{G}'(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O}' : (\,|\,\mathsf{G}(\Theta) \vdash \neg A : \mathcal{N})\end{array} \end{array}$$

Shift Laws

$$\begin{array}{rcl} \begin{array}{l}\textbf{codata}\,\mathsf{G}(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{K} : (\,|\,\mathsf{G}(\Theta) \vdash A : \mathcal{S})\end{array} & \approx_{\uparrow_{\mathcal{S}} R} & \begin{array}{l}\textbf{codata}\,\mathsf{G}'(\Theta) : \mathcal{N}\,\textbf{where}\\ \quad \mathsf{O}' : (\,|\,\mathsf{G}'(\Theta) \vdash {\uparrow_{\mathcal{S}}} A : \mathcal{N})\end{array} \end{array}$$

FIGURE 8.7. Isomorphism laws of negatively polarized co-data sub-structures.

the negating $\sim L$ law, where we know that $\mathsf{F}(\Theta)\theta \approx \sim A\theta$ by Lemma 8.3 (b) because $A\theta \approx A\theta$ by reflexivity, and as usual $\mathsf{F}'(\Theta)\theta \approx \sim A\theta$ by Lemma 8.5 (a). Additionally, the shifting $\downarrow_{\mathcal{S}} L$ law is sound because we know that $\mathsf{F}(\Theta)\theta \approx \downarrow_{\mathcal{S}} A\theta$ by Lemma 8.3 (a) because of the reflexive isomorphism $A\theta \approx A\theta$, and $\mathsf{F}'(\Theta)\theta \approx \downarrow_{\mathcal{S}} A\theta$ by Lemma 8.5 (a).

And finally, the sub-structural laws for the binary (co-)data types $(\oplus, \otimes)$ require the most effort. This is because each of these types have two parts, and so we must relate one part at a time and then mix the result together. In particular, we know that $\mathsf{F}_1(\Theta)\theta \approx \mathsf{F}_1'(A, B)\theta$ and $\mathsf{F}_2(\Theta)\theta \approx \mathsf{F}_2'(A, B)\theta$ for the declarations

$$\begin{array}{ll}
\mathbf{data}\, \mathsf{F}_1(\Theta) : \mathcal{V}\, \mathbf{where} & \mathbf{data}\, \mathsf{F}_1'(X : \mathcal{V}, Y : \mathcal{V}) : \mathcal{V}\, \mathbf{where} \\
\quad \mathsf{K}_1 : (A : \mathcal{V} \vdash \mathsf{F}_1(\Theta) \mid\,) & \quad \mathsf{K}_1' : (X : \mathcal{V} \vdash \mathsf{F}_1'(\Theta) \mid\,) \\
\mathbf{data}\, \mathsf{F}_2(\Theta) : \mathcal{V}\, \mathbf{where} & \mathbf{data}\, \mathsf{F}_2'(X : \mathcal{V}, Y : \mathcal{V}) : \mathcal{V}\, \mathbf{where} \\
\quad \mathsf{K}_2 : (B : \mathcal{V} \vdash \mathsf{F}_2(\Theta) \mid\,) & \quad \mathsf{K}_2' : (Y : \mathcal{V} \vdash \mathsf{F}_2'(\Theta) \mid\,)
\end{array}$$

by applying Lemma 8.3 (a) to the reflexive isomorphisms $A\theta \approx X\{A\theta/X\}$ and $B\theta \approx Y\{B\theta/Y\}$. Now note the two different ways to mix these isomorphisms together with Lemma 8.2. First, we could mix the above $\mathsf{F}_1(\Theta)\theta \approx \mathsf{F}_1'(A, B)\theta$ and $\mathsf{F}_2(\Theta)\theta \approx \mathsf{F}_2'(A, B)\theta$ as the first two isomorphisms while the third is $\mathsf{F}_3(\Theta)\theta \approx \mathsf{F}_3'(A, B)\theta$ given by Lemma 8.1 (a) of the trivial data declarations

$$\begin{array}{ll}
\mathbf{data}\, \mathsf{F}_3(\Theta) : \mathcal{V}\, \mathbf{where} & \mathbf{data}\, \mathsf{F}_3'(X : \mathcal{V}, Y : \mathcal{V}) : \mathcal{V}\, \mathbf{where} \\
\quad \mathsf{K}_3 : (\, \vdash \mathsf{F}_3(\Theta) \mid\,) & \quad \mathsf{K}_3' : (\, \vdash \mathsf{F}_3'(X, Y) \mid\,)
\end{array}$$

which tells us that $\mathsf{F}(\Theta)\theta \approx A \oplus B$ as required by the $\oplus L$ law. Second, we could mix $\mathsf{F}_1(\Theta)\theta \approx \mathsf{F}_1'(A, B)\theta$ and $\mathsf{F}_2(\Theta)\theta \approx \mathsf{F}_2'(A, B)\theta$ as the second two isomorphisms while the first is $\mathsf{F}_0(\Theta)\theta \approx \mathsf{F}_0'(A, B)\theta$ by Lemma 8.1 (b) of the trivial data declarations $\mathbf{data}\, \mathsf{F}_3(\Theta) : \mathcal{V}\, \mathbf{where}$ and $\mathbf{data}\, \mathsf{F}_3'(X : \mathcal{V}, Y : \mathcal{V}) : \mathcal{V}\, \mathbf{where}$ which tells us that $\mathsf{F}(\Theta)\theta \approx A \otimes B$ as required by the $\otimes L$ law. □

In addition to the specific laws of Figures 8.6 and 8.7, each of the polarized connectives is compatible with isomorphism. For example, if we have $A \approx A'$, then we also have $A \oplus B \approx A' \oplus B$ and $B \oplus A \approx B \oplus A'$. This fact lets us apply type isomorphisms within the context of certain larger types: if two types are isomorphic, then we can build on them with polarized connectives however we want and still have an isomorphism. Said another way, for any type A made from polarized connectives and any other isomorphic types $B \approx C$, we can substitute both B and C for X in A and still have the isomorphism $A\{B/X\} \approx A\{C/X\}$.

Theorem 8.8 (Polarized isomorphism substitution)**.** *For any* $\Theta, X : \mathcal{S} \vdash_{\mathcal{P}} A : \mathcal{T}$*,* $\Theta \vdash_{\mathcal{G}} B : \mathcal{S}$*, and* $\Theta \vdash_{\mathcal{G}} C : \mathcal{S}$*, if* $\Theta \vDash B \approx C$ *then* $\Theta \vDash A\{B/X\} \approx A\{C/X\}$*.*

Proof. By induction on the typing derivation of $\Theta, X : \mathcal{S} \vdash_{\mathcal{P}} A : \mathcal{T}$ and the fact that each polarized connective is compatible with isomorphism. For example, in the case of $\otimes$, given that $A_1 \approx A'_1$ and $A_2 \approx A'_2$ we have that

$$\begin{array}{rcl}
\begin{array}{l}\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\\ \quad \mathsf{K} : (A_1 \otimes A_2 : \mathcal{V} \vdash \mathsf{F}_1() \mid)\end{array} & \approx_{\otimes L} & \begin{array}{l}\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\\ \quad \mathsf{K} : (A_1 : \mathcal{V}, A_2 : \mathcal{V} \vdash \mathsf{F}_2() \mid)\end{array} \\
 & \approx & \begin{array}{l}\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\\ \quad \mathsf{K} : (A'_1 : \mathcal{V}, A'_2 : \mathcal{V} \vdash \mathsf{F}_3() \mid)\end{array} \\
 & \approx_{\otimes L} & \begin{array}{l}\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\\ \quad \mathsf{K} : (A'_1 \otimes A'_2 : \mathcal{V} \vdash \mathsf{F}_4() \mid)\end{array}
\end{array}$$

by the $\otimes L$, data compatibility, and data mix laws, and so $A_1 \otimes A_2 \approx \mathsf{F}_1() \approx \mathsf{F}_4() \approx A'_1 \otimes A'_2$ by Lemma 8.5. The compatibility of the other polarized connectives follows similarly. □

Laws of the Polarized Basis

We have just seen in the previous section that there is *an* encoding of user-defined (co-)data types solely in terms of the basic polarized connectives. However, how do we know that this encoding is canonical, or that there are not many different and unrelated encodings for the same purpose? Does it matter what order in which the components of (co-)data types are put together, or in which way they are nested? Or maybe we could instead encode (co-)data types in terms of the positive $\oplus$ and $\otimes$ connectives instead of the negative & and ⅋?

As it turns out, none of these differences matter. The advantage of using the polarized connectives, as declared in Figure 8.1, as the basis for encodings is that they exhibit many pleasant—if none too surprising—properties, some of which have been explored previously by Zeilberger (2009) and Munch-Maccagnoni (2013). That is, in contrast with types like call-by-name tuples or call-by-value functions, the relationships between types that we should expect—corresponding to common and well-known

$$
\begin{array}{cc}
A \oplus B \approx B \oplus A & A \mathbin{\&} B \approx B \mathbin{\&} A \\
(A \oplus B) \oplus C \approx A \oplus (B \oplus C) & (A \mathbin{\&} B) \mathbin{\&} C \approx A \mathbin{\&} (B \mathbin{\&} C) \\
0 \oplus A \approx A \approx A \oplus 0 & \top \mathbin{\&} A \approx A \approx A \mathbin{\&} \top \\[1ex]
A \otimes B \approx B \otimes A & A \parr B \approx B \parr A \\
(A \otimes B) \otimes C \approx A \otimes (B \otimes C) & (A \parr B) \parr C \approx A \parr (B \parr C) \\
1 \otimes A \approx A \approx A \otimes 1 & \bot \parr A \approx A \approx A \parr \bot \\[1ex]
A \otimes (B \oplus C) \approx (A \otimes B) \oplus (A \otimes C) & A \parr (B \mathbin{\&} C) \approx (A \parr B) \mathbin{\&} (A \parr C) \\
(A \oplus B) \otimes C \approx (A \otimes C) \oplus (B \otimes C) & (A \mathbin{\&} B) \parr C \approx (A \parr C) \mathbin{\&} (B \parr C) \\
A \otimes 0 \approx 0 \approx 0 \otimes A & A \parr \top \approx \top \approx \top \parr A
\end{array}
$$

FIGURE 8.8. Algebraic laws of the polarized basis of types.

relationships from algebra and logic—are actually isomorphisms between polarized types even in the face of effects that let terms avoid giving a result.

Algebraic laws

Let's begin by first exploring the algebraic properties of the polarized connectives. In particular, the isomorphic relationship between the additive and multiplicative connectives from Figure 8.1

- On the positive side, the $\oplus$ and 0 connectives form a *commutative monoid* of types up to isomorphism—meaning they satisfy commutative, associative, and unit laws as isomorphisms between types—and so do the $\otimes$ and 1 connectives. Furthermore, all four together form a *commutative semiring* up to isomorphism—meaning the "multiplication" $\otimes$ distributes over the "addition" $\oplus$ and is annihilated by the "zero" 0.

- On the negative side, the & and $\top$ connectives form a commutative monoid up to isomorphism and $\parr$ and $\bot$ do as well. All four together form a commutative semiring with & as addition and $\parr$ as mutiplication.

These properties of the additive and multiplicative connectives are summarized in Figure 8.8.

We can verify that each of these isomorphisms are, in fact, isomorphisms using the previously-established laws of (co-)data declarations in general and internal polarizedsubstructures in particular from Figures 8.4, 8.5, 8.6 and 8.7. The general technique follows the observation that, because of Lemma 8.5, if we have either a singleton data declaration isomorphism or a singleton (co-)data declaration isomorphism of the form:

$$\begin{array}{r} \mathbf{data}\,\mathsf{F}() : \mathcal{V}\,\mathbf{where} \\ \mathsf{K} : (A : \mathcal{V} \vdash \mathsf{F}() \mid\,) \end{array} \approx \begin{array}{l} \mathbf{data}\,\mathsf{F}'() : \mathcal{V}\,\mathbf{where} \\ \quad \mathsf{K}' : (A' : \mathcal{V} \vdash \mathsf{F}'() \mid\,) \end{array}$$

or

$$\begin{array}{r} \mathbf{codata}\,\mathsf{G}() : \mathcal{N}\,\mathbf{where} \\ \mathsf{O} : (\,\mid \mathsf{G}() \vdash A : \mathcal{N}) \end{array} \approx \begin{array}{l} \mathbf{codata}\,\mathsf{G}'() : \mathcal{N}\,\mathbf{where} \\ \quad \mathsf{O}' : (\,\mid \mathsf{G}'() \vdash A' : \mathcal{N}) \end{array}$$

then we have $A \approx A'$ by composing $A \approx \mathsf{F}() \approx \mathsf{F}'() \approx A'$ or $A \approx \mathsf{G}() \approx \mathsf{G}'() \approx A'$. Therefore, we can prove isomorphism laws about the polarized (co-)data types by (1) placing both sides of the proposed isomorphism within a singleton data or co-data type, as appropriate, (2) "unpacking" the two sides within the structure of the containing (co-)data type declaration, and (3) use the laws of declaration isomorphisms to show the two sides are indeed isomorphic. Each of these algebraic laws can be derived from the laws in Figures 8.6 and 8.7 as follows.

Commutativity

The commutativity laws for reordering the binary connectives, unsurprisingly, follows from the commutativity laws for reordering the parts of declarations. For the multiplicative $\otimes$ and $⅋$, we use the first commute law to reorder the components within a single constructor or observer, as follows:

$$\begin{aligned} &\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \otimes B : \mathcal{V} \vdash \mathsf{F}_1() \mid\,) \\ &\approx_{\otimes L} \mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V}, B : \mathcal{V} \vdash \mathsf{F}_2() \mid\,) \\ &\approx \mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (B : \mathcal{V}, A : \mathcal{V} \vdash \mathsf{F}_3() \mid\,) \\ &\approx_{\otimes L} \mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (B \otimes A : \mathcal{V} \vdash \mathsf{F}_4() \mid\,) \end{aligned}$$

$$\begin{aligned} &\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_1() \vdash A ⅋ B : \mathcal{N}) \\ &\approx_{⅋ R} \mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_2() \vdash A : \mathcal{N}, B : \mathcal{N}) \\ &\approx \mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_3() \vdash B : \mathcal{N}, A : \mathcal{N}) \\ &\approx_{⅋ L} \mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_4() \vdash B ⅋ A : \mathcal{N}) \end{aligned}$$

Whereas for the additive $\oplus$ and $\&$, we use the second commute law to reorder the alternatives within a declaration as shown in the following isomorphism:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \oplus B : \mathcal{V} \vdash \mathsf{F}_1() \mid)\\
\approx_{\oplus L}\ &\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (A : \mathcal{V} \vdash \mathsf{F}_2() \mid)\\
&\qquad\qquad\qquad\qquad\ \ \mathsf{K}_2 : (B : \mathcal{V} \vdash \mathsf{F}_2() \mid)\\
\approx\ &\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_2 : (B : \mathcal{V} \vdash \mathsf{F}_3() \mid)\\
&\qquad\qquad\qquad\qquad\ \ \mathsf{K}_1 : (A : \mathcal{V} \vdash \mathsf{F}_3() \mid)\\
\approx_{\oplus L}\ &\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (B \oplus A : \mathcal{V} \vdash \mathsf{F}_4() \mid)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_1() \vdash A \,\&\, B : \mathcal{N})\\
\approx_{\& R}\ &\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\mid \mathsf{G}_2() \vdash A : \mathcal{N})\\
&\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{O}_2 : (\mid \mathsf{G}_2() \vdash B : \mathcal{N})\\
\approx\ &\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_2 : (\mid \mathsf{G}_3() \vdash B : \mathcal{N})\\
&\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{O}_1 : (\mid \mathsf{G}_3() \vdash A : \mathcal{N})\\
\approx_{\& R}\ &\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_4() \vdash B \,\&\, A : \mathcal{N})
\end{aligned}$$

Unit

Combining the binary connectives with their corresponding units is an identity operation that leaves types unchanged, up to isomorphism. These unit laws rely on the fact that the right and left laws for the nullary connectives "cancel out," in an appropriate way, any occurence of the nullary connective within a (co-)data declaration as described by the $1L$, $0L$, $\bot R$, and $\top R$ laws. For the multiplicative 1 and $\bot$ connectives, we use the fact that 1 vanishes from the left-hand side of a constructor and $\bot$ vanishes from the right-hand side of an observer:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (1 \otimes A : \mathcal{V} \vdash \mathsf{F}_1() \mid)\\
\approx_{\otimes L}\ &\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (1 : \mathcal{V}, A : \mathcal{V} \vdash \mathsf{F}_2() \mid)\\
\approx_{1L}\ &\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V} \vdash \mathsf{F}_3() \mid)\\
\approx_{1L}\ &\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V}, 1 : \mathcal{V} \vdash \mathsf{F}_4() \mid)\\
\approx_{\otimes L}\ &\mathbf{data}\,\mathsf{F}_5() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \otimes 1 : \mathcal{V} \vdash \mathsf{F}_5() \mid)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\;|\;\mathsf{G}_1() \vdash \bot \parr A : \mathcal{N})\\
\approx_{\parr R}\;&\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\;|\;\mathsf{G}_2() \vdash \bot : \mathcal{N}, A : \mathcal{N})\\
\approx_{\bot R}\;&\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\;|\;\mathsf{G}_3() \vdash A : \mathcal{N})\\
\approx_{\bot R}\;&\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\;|\;\mathsf{G}_4() \vdash A : \mathcal{N}, \bot : \mathcal{N})\\
\approx_{\parr R}\;&\mathbf{codata}\,\mathsf{G}_5() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\;|\;\mathsf{G}_5() \vdash A \parr \bot : \mathcal{N})
\end{aligned}$$

Note the use of the mix law to extend $1L$ and $\bot R$ to allow for an extra component along side the unit connective. Alternatively, for the additive 0 and $\top$ connectives, we use the fact that any constructor containing a 0 on its left-hand side completely vanishes itself, whereas an observer containing a $\top$ on its right-hand side vanishes:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (0 \oplus A : \mathcal{V} \vdash \mathsf{F}_1() \mid\;)\\
\approx_{\oplus L}\;&\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (0 : \mathcal{V} \vdash \mathsf{F}_2() \mid\;)\\
&\phantom{\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_2 : (A : \mathcal{V} \vdash \mathsf{F}_2() \mid\;)\\
\approx_{0L}\;&\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V} \vdash \mathsf{F}_3() \mid\;)\\
\approx_{0L}\;&\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (A : \mathcal{V} \vdash \mathsf{F}_4() \mid\;)\\
&\phantom{\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_2 : (0 : \mathcal{V} \vdash \mathsf{F}_4() \mid\;)\\
\approx_{\oplus L}\;&\mathbf{data}\,\mathsf{F}_5() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \oplus 0 : \mathcal{V} \vdash \mathsf{F}_5() \mid\;)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\;|\;\mathsf{G}_1() \vdash \top \,\&\, A : \mathcal{N})\\
\approx_{\& R}\;&\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\;|\;\mathsf{G}_2() \vdash \top : \mathcal{N})\\
&\phantom{\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,}\mathsf{O}_2 : (\;|\;\mathsf{G}_2() \vdash A : \mathcal{N})\\
\approx_{\top R}\;&\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\;|\;\mathsf{G}_3() \vdash A : \mathcal{N})\\
\approx_{\top R}\;&\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\;|\;\mathsf{G}_4() \vdash A : \mathcal{N})\\
&\phantom{\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,}\mathsf{O}_2 : (\;|\;\mathsf{G}_4() \vdash \top : \mathcal{N})\\
\approx_{\& R}\;&\mathbf{codata}\,\mathsf{G}_5() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\;|\;\mathsf{G}_5() \vdash A \,\&\, \top : \mathcal{N})
\end{aligned}$$

Again, the mix law is used to extend $0L$ and $\top R$ for (co-)data declarations with another alternative.

Associativity

Nested applications of the same binary connective can be reassociated, up to isomorphism. This is because (co-)data declarations are "flat:" there is a single, flat list of alternative, with each one containing a single, flat list of components on either side of the turnstyle. Therefore, after we fully unpack a nested application of a connective, it flattens out, so that we may repack the same parts back together in the other order. For the multiplicative $\otimes$ and $⅋$, we have the following isomorphism:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : ((A \otimes B) \otimes C : \mathcal{V} \vdash \mathsf{F}_1() \mid)\\
\approx_{\otimes L}\ &\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \otimes B : \mathcal{V}, C : \mathcal{V} \vdash \mathsf{F}_2() \mid)\\
\approx_{\otimes L}\ &\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V}, B : \mathcal{V}, C : \mathcal{V} \vdash \mathsf{F}_3() \mid)\\
\approx_{\otimes L}\ &\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V}, B \otimes C : \mathcal{V} \vdash \mathsf{F}_4() \mid)\\
\approx_{\otimes L}\ &\mathbf{data}\,\mathsf{F}_5() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \otimes (B \otimes C) : \mathcal{V} \vdash \mathsf{F}_5() \mid)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_1() \vdash (A ⅋ B) ⅋ C : \mathcal{N})\\
\approx_{⅋R}\ &\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_2() \vdash A ⅋ B : \mathcal{N}, C : \mathcal{N})\\
\approx_{⅋R}\ &\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_3() \vdash A : \mathcal{N}, B : \mathcal{N}, C : \mathcal{N})\\
\approx_{⅋R}\ &\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_4() \vdash A : \mathcal{N}, B ⅋ C : \mathcal{N})\\
\approx_{⅋R}\ &\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_4() \vdash A ⅋ (B ⅋ C) : \mathcal{N})
\end{aligned}$$

Note that the mix law is used to extend $\otimes L$ and $⅋R$ to allow for an extra component on either side of the main pair. For the additive $\oplus$ and $\&$, we have the following isomorphisms, again using mix to extend $\oplus L$ and $\& R$ to allow for an extra alternative before or after the main pair:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : ((A \oplus B) \oplus C : \mathcal{V} \vdash \mathsf{F}_1() \mid) \\
\approx_{\oplus L}\; &\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (A \oplus B : \mathcal{V} \vdash \mathsf{F}_2() \mid) \\
&\phantom{\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_2 : (C : \mathcal{V} \vdash \mathsf{F}_2() \mid) \\
\approx_{\oplus L}\; &\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (A : \mathcal{V} \vdash \mathsf{F}_3() \mid) \\
&\phantom{\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_2 : (B : \mathcal{V} \vdash \mathsf{F}_3() \mid) \\
&\phantom{\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_3 : (C : \mathcal{V} \vdash \mathsf{F}_3() \mid) \\
\approx_{\oplus L}\; &\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (A : \mathcal{V} \vdash \mathsf{F}_4() \mid) \\
&\phantom{\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_2 : (B \oplus C : \mathcal{V} \vdash \mathsf{F}_4() \mid) \\
\approx_{\oplus L}\; &\mathbf{data}\,\mathsf{F}_5() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \oplus (B \oplus C) : \mathcal{V} \vdash \mathsf{F}_5() \mid)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_1() \vdash (A \,\&\, B) \,\&\, C : \mathcal{N}) \\
\approx_{\& R}\; &\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\mid \mathsf{G}_2() \vdash A \,\&\, B : \mathcal{N}) \\
&\phantom{\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,}\mathsf{O}_2 : (\mid \mathsf{G}_2() \vdash C : \mathcal{N}) \\
\approx_{\& R}\; &\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\mid \mathsf{G}_3() \vdash A : \mathcal{N}) \\
&\phantom{\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,}\mathsf{O}_2 : (\mid \mathsf{G}_3() \vdash B : \mathcal{N}) \\
&\phantom{\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,}\mathsf{O}_3 : (\mid \mathsf{G}_3() \vdash C : \mathcal{N}) \\
\approx_{\& R}\; &\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\mid \mathsf{G}_4() \vdash A : \mathcal{N}) \\
&\phantom{\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,}\mathsf{O}_2 : (\mid \mathsf{G}_4() \vdash B \,\&\, C : \mathcal{N}) \\
\approx_{\& R}\; &\mathbf{codata}\,\mathsf{G}_5() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_5() \vdash A \,\&\, (B \,\&\, C) : \mathcal{N})
\end{aligned}$$

Distributivity

Distributing a multiplication over an addition also arises from the flat nature of (co-)data declarations much like reassociating a binary connective. The difference is that when the addition is flattened out into the structure of the declaration, the multiplied type is carried along for the ride (via the mix law) and copied across both alternatives, as shown in the following isomorphisms:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \otimes (B \oplus C) : \mathcal{V} \vdash \mathsf{F}_1() \mid)\\
\approx_{\otimes L}\;&\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V}, B \oplus C : \mathcal{V} \vdash \mathsf{F}_2() \mid)\\
\approx_{\oplus L}\;&\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (A : \mathcal{V}, B : \mathcal{V} \vdash \mathsf{F}_3() \mid)\\
&\qquad\qquad\qquad\qquad\quad\;\, \mathsf{K}_2 : (A : \mathcal{V}, C : \mathcal{V} \vdash \mathsf{F}_3() \mid)\\
\approx_{\otimes L}\;&\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (A \otimes B : \mathcal{V} \vdash \mathsf{F}_4() \mid)\\
&\qquad\qquad\qquad\qquad\quad\;\, \mathsf{K}_2 : (A : \mathcal{V}, C : \mathcal{V} \vdash \mathsf{F}_4() \mid)\\
\approx_{\otimes L}\;&\mathbf{data}\,\mathsf{F}_5() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (A \otimes B : \mathcal{V} \vdash \mathsf{F}_5() \mid)\\
&\qquad\qquad\qquad\qquad\quad\;\, \mathsf{K}_2 : (A \otimes C : \mathcal{V} \vdash \mathsf{F}_5() \mid)\\
\approx_{\oplus L}\;&\begin{array}{l}\mathbf{data}\,\mathsf{F}_6() : \mathcal{V}\,\mathbf{where}\\ \quad \mathsf{K} : ((A \otimes B) \oplus (A \otimes C) : \mathcal{V} \vdash \mathsf{F}_6() \mid)\end{array}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_1() \vdash A \parr (B \mathbin{\&} C) : \mathcal{N})\\
\approx_{\parr R}\;&\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_2() \vdash A : \mathcal{N}, B \mathbin{\&} C : \mathcal{N})\\
\approx_{\& R}\;&\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\mid \mathsf{G}_3() \vdash A : \mathcal{N}, B : \mathcal{N})\\
&\qquad\qquad\qquad\qquad\qquad\;\, \mathsf{O}_2 : (\mid \mathsf{G}_3() \vdash A : \mathcal{N}, C : \mathcal{N})\\
\approx_{\parr R}\;&\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\mid \mathsf{G}_4() \vdash A \parr B : \mathcal{N})\\
&\qquad\qquad\qquad\qquad\qquad\;\, \mathsf{O}_2 : (\mid \mathsf{G}_4() \vdash A : \mathcal{N}, C : \mathcal{N})\\
\approx_{\parr R}\;&\mathbf{codata}\,\mathsf{G}_5() : \mathcal{N}\,\mathbf{where}\,\mathsf{O}_1 : (\mid \mathsf{G}_5() \vdash A \parr B : \mathcal{N})\\
&\qquad\qquad\qquad\qquad\qquad\;\, \mathsf{O}_2 : (\mid \mathsf{G}_5() \vdash A \parr C : \mathcal{N})\\
\approx_{\& R}\;&\begin{array}{l}\mathbf{codata}\,\mathsf{G}_6() : \mathcal{N}\,\mathbf{where}\\ \quad \mathsf{O} : (\mid \mathsf{G}_6() \vdash (A \parr B) \mathbin{\&} (A \parr Cx) : \mathcal{N})\end{array}
\end{aligned}$$

Annihilation

When a type is multiplied by the additive unit, it is cancelled out. This occurs because, unlike an addition, the multiplication places the type next to the unit where it is in harms way. Thus, when $0L$ and $\top R$ are extended (by the mix law) to allow for an extra component alongside the units, it is swept aside as the entire alternative is deleted, as in the following isomorphisms:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A \otimes 0 : \mathcal{V} \vdash \mathsf{F}_1() \mid\,) \\
\approx_{\otimes L}\ &\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V}, 0 : \mathcal{V} \vdash \mathsf{F}_2() \mid\,) \\
\approx_{0L}\ &\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where} \\
\approx_{0L}\ &\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (0 : \mathcal{V}, A : \mathcal{V} \vdash \mathsf{F}_4() \mid\,) \\
\approx_{\otimes L}\ &\mathbf{data}\,\mathsf{F}_5() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (0 \otimes A : \mathcal{V} \vdash \mathsf{F}_5() \mid\,)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\, \mid \mathsf{G}_1() \vdash A \parr \top : \mathcal{N}) \\
\approx_{\parr R}\ &\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\, \mid \mathsf{G}_2() \vdash A : \mathcal{N}, \top : \mathcal{N}) \\
\approx_{\top R}\ &\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where} \\
\approx_{\top R}\ &\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\, \mid \mathsf{G}_4() \vdash \top : \mathcal{N}, A : \mathcal{N}) \\
\approx_{\parr R}\ &\mathbf{codata}\,\mathsf{G}_5() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\, \mid \mathsf{G}_5() \vdash \top \parr A : \mathcal{N})
\end{aligned}$$

Duality laws

Isomorphism of types also gives us common logical properties of the polarized connectives based on duality, established with the same technique used in Section 8.4. In particular, we get two parallel copies of the De Morgan laws—one for $\sim$ negation and the other for $\neg$ negation—that relates the positive data types with the negative co-data types as shown in Figure 8.9. The positive "or" ($\oplus$) is dualized into the negative "and" ($\&$) and the positive "and" ($\otimes$) is dualized into the negative "or" ($\parr$). Additionally, the two negation connectives cancel each other out, up to isomorphism. That is to say, they are a characterization of *involutive negation* as (co-)data types.[5]

Remark 8.1. Note that, while the polarized basis for (co-)data types from Figure 8.1 is nicely symmetric, it is also somewhat redundant. The fact that the negation connectives are involutive and follow the de Morgan laws means that we get the following derived type isomorphisms:

$$\begin{aligned}
A \oplus B &\approx {\sim}((\neg A) \,\&\, (\neg B)) & \qquad A \,\&\, B &\approx \neg(({\sim}A) \oplus ({\sim}B)) \\
A \otimes B &\approx {\sim}((\neg A) \parr (\neg B)) & \qquad A \parr B &\approx \neg(({\sim}A) \otimes ({\sim}B))
\end{aligned}$$

[5] This fact was noticed by Zeilberger (2009) and further brought to the forefront by Munch-Maccagnoni (2014). The key is to have two dual negations, where one can be encoded with implication ($\neg A \approx A \to \bot$) and its dual can be encoded with subtraction (${\sim}A \approx 1 - A$).

$$\begin{aligned}
{\sim}(A \mathbin{\&} B) &\approx ({\sim}A) \oplus ({\sim}B) & \qquad \neg(A \oplus B) &\approx (\neg A) \mathbin{\&} (\neg B) \\
{\sim}\top &\approx 0 & \neg 0 &\approx \top \\
{\sim}(A \parr B) &\approx ({\sim}A) \otimes ({\sim}B) & \neg(A \otimes B) &\approx (\neg A) \parr (\neg B) \\
{\sim}\bot &\approx 1 & \neg 1 &\approx \bot \\
{\sim}(\neg A) &\approx A & \neg({\sim}A) &\approx A
\end{aligned}$$

FIGURE 8.9. De Morgan duality laws of the polarized basis of types.

The consequence of these isomorphisms is that we only really need *half* the listed additive and multiplicative connectives. We could, for example, take only $\oplus$, 0, $\otimes$, 1, $\sim$, and $\neg$ and primitive connectives and encode $\&$, $\top$, $\parr$, and $\bot$ in terms of them as above. Dually, we could take $\&$, $\top$, $\parr$, $\bot$, $\sim$, and $\neg$ as primitive and encode $\oplus$, 0, $\otimes$, 1. Or we could instead mix and match between the positive (data) and negative (co-data) forms as desired. The only requirement is that we have at least *one* binary and nullary additive connective, *one* binary and nullary multiplicative connective, and *both* dual involutive negations. *End remark* 8.1.

Involutive negation

Double negation elimination (both the positive ${\sim}(\neg A) \approx A$ and negative $\neg({\sim}A) \approx A$ forms) is perhaps deceptively simple: the ${\sim}L$ and $\neg R$ laws just flip the double-negated type back and forth across the turnstyle until both negations disappear:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : ({\sim}(\neg A) : \mathcal{V} \vdash \mathsf{F}_1 \mid\ ) \\
\approx_{{\sim}L}\ &\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (\ \vdash \mathsf{F}_2 \mid \neg A : \mathcal{N}) \\
\approx_{\neg R}\ &\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (A : \mathcal{V} \vdash \mathsf{F}_3 \mid\ )
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{O} : (\ \mid \mathsf{G}_1 \vdash \neg({\sim}A) : \mathcal{N}) \\
\approx_{\neg R}\ &\mathbf{codata}\,\mathsf{G}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{O} : ({\sim}A : \mathcal{V} \mid \mathsf{G}_2 \vdash\ ) \\
\approx_{{\sim}L}\ &\mathbf{codata}\,\mathsf{G}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{O} : (\ \mid \mathsf{G}_3 \vdash A : \mathcal{N})
\end{aligned}$$

Note that in the case of ${\sim}(\neg A)$, $\neg R$ must be used in a data declaration instead of a co-data declaration, and likewise ${\sim}L$ must be used in a co-data declaration for $\neg({\sim}A)$. This can be accomplished with the (co-)data interchange laws, that let us convert each data isomorphism from Figures 8.6 and 8.6 into a co-data isomorphism and vice versa.

Constant negation

Involutive negation converts "true" into "false" and "false" into "true," but it also swaps between the data and co-data formulations of each. For the multiplicative units, the data type 1 for true is the negation of the co-data type $\bot$ for false because both represent a (co-)data type with one alternative containing nothing:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : ({\sim}\bot : \mathcal{V} \vdash \mathsf{F}_1 \mid\,) \\
&\approx_{\sim L}\ \mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (\,\vdash \mathsf{F}_2 \mid \bot : \mathcal{N}) \\
&\approx_{\bot R}\ \mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (\,\vdash \mathsf{F}_3 \mid\,) \\
&\approx_{1L}\ \mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (1 : \mathcal{V} \vdash \mathsf{F}_4 \mid\,)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_1 \vdash \neg 1 : \mathcal{N}) \\
&\approx_{\neg R}\ \mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (1 : \mathcal{V} \mid \mathsf{G}_2 \vdash\,) \\
&\approx_{1L}\ \mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_3 \vdash\,) \\
&\approx_{\bot R}\ \mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_4 \vdash \bot : \mathcal{N})
\end{aligned}$$

For the additive units, the data type 0 for false is the negation of the co-data type $\top$ for true because both represent a (co-)data type with no alternatives:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : ({\sim}\top : \mathcal{V} \vdash \mathsf{F}_1() \mid\,) \\
&\approx_{\sim L}\ \mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (\,\vdash \mathsf{F}_2() \mid \top : \mathcal{N}) \\
&\approx_{\top R}\ \mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where} \\
&\approx_{0L}\ \mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (0 : \mathcal{V} \vdash \mathsf{F}_4() \mid\,)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_1() \vdash \neg 0 : \mathcal{N}) \\
&\approx_{\neg R}\ \mathbf{codata}\,\mathsf{G}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{O} : (0 : \mathcal{V} \mid \mathsf{G}_2() \vdash\,) \\
&\approx_{0L}\ \mathbf{codata}\,\mathsf{G}_3() : \mathcal{V}\,\mathbf{where} \\
&\approx_{\top R}\ \mathbf{codata}\,\mathsf{G}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{O} : (\,\mid \mathsf{G}_4() \vdash \top : \mathcal{N})
\end{aligned}$$

De Morgan laws

Involutive negation also converts “and” into “or” and “or” into “and” while interchanging data with co-data. For the multiplicatives, the connective $\otimes$ is an “and” pair that amalgamates two pieces of data into a single structure, and this is the negation of $\parr$ which is an “or” pair that conjoins two observers together:

$$\begin{aligned}
&\mathbf{data}\, \mathsf{F}_1() : \mathcal{V}\, \mathbf{where}\, \mathsf{K} : ({\sim}(A \parr B) : \mathcal{V} \vdash \mathsf{F}_1() \mid\,) \\
\approx_{\sim L}\ &\mathbf{data}\, \mathsf{F}_2() : \mathcal{V}\, \mathbf{where}\, \mathsf{K} : (\, \vdash \mathsf{F}_2() \mid A \parr B : \mathcal{N}) \\
\approx_{\parr R}\ &\mathbf{data}\, \mathsf{F}_3() : \mathcal{V}\, \mathbf{where}\, \mathsf{K} : (\, \vdash \mathsf{F}_3() \mid A : \mathcal{N}, B : \mathcal{N}) \\
\approx_{\sim L}\ &\mathbf{data}\, \mathsf{F}_4() : \mathcal{V}\, \mathbf{where}\, \mathsf{K} : ({\sim}A : \mathcal{V} \vdash \mathsf{F}_4() \mid B : \mathcal{N}) \\
\approx_{\sim L}\ &\mathbf{data}\, \mathsf{F}_5() : \mathcal{V}\, \mathbf{where}\, \mathsf{K} : ({\sim}A : \mathcal{V}, {\sim}B : \mathcal{V} \vdash \mathsf{F}_5() \mid\,) \\
\approx_{\otimes L}\ &\mathbf{data}\, \mathsf{F}_6() : \mathcal{V}\, \mathbf{where}\, \mathsf{K} : (({\sim}A) \otimes ({\sim}B) : \mathcal{V} \vdash \mathsf{F}_6() \mid\,)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\, \mathsf{G}_1() : \mathcal{N}\, \mathbf{where}\, \mathsf{O} : (\, \mid \mathsf{G}_1() \vdash \neg(A \otimes B) : \mathcal{N}) \\
\approx_{\neg R}\ &\mathbf{codata}\, \mathsf{G}_2() : \mathcal{N}\, \mathbf{where}\, \mathsf{O} : (A \otimes B : \mathcal{V} \mid \mathsf{G}_2() \vdash\,) \\
\approx_{\otimes L}\ &\mathbf{codata}\, \mathsf{G}_3() : \mathcal{N}\, \mathbf{where}\, \mathsf{O} : (A : \mathcal{V}, B : \mathcal{V} \mid \mathsf{G}_3() \vdash\,) \\
\approx_{\neg R}\ &\mathbf{codata}\, \mathsf{G}_4() : \mathcal{N}\, \mathbf{where}\, \mathsf{O} : (A : \mathcal{V} \mid \mathsf{G}_4() \vdash \neg B : \mathcal{N}) \\
\approx_{\neg R}\ &\mathbf{codata}\, \mathsf{G}_5() : \mathcal{N}\, \mathbf{where}\, \mathsf{O} : (\, \mid \mathsf{G}_5() \vdash \neg A : \mathcal{N}, \neg B : \mathcal{N}) \\
\approx_{\parr R}\ &\mathbf{codata}\, \mathsf{G}_6() : \mathcal{N}\, \mathbf{where}\, \mathsf{O} : (\, \mid \mathsf{G}_6() \vdash (\neg A) \parr (\neg B) : \mathcal{N})
\end{aligned}$$

For the additives, the connective $\oplus$ is an “or” that yields one of two possible alternative types of answers, and this is the negation of & which gives observers the option of one of two possible types of questions:

$$\begin{aligned}
&\mathbf{data}\,\mathsf{F}_1() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : ({\sim}(A \mathbin{\&} B) : \mathcal{V} \vdash \mathsf{F}_1() \mid)\\
\approx_{\sim L}\ &\mathbf{data}\,\mathsf{F}_2() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (\vdash \mathsf{F}_2() \mid A \mathbin{\&} B : \mathcal{N})\\
\approx_{\& R}\ &\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : (\vdash \mathsf{F}_3() \mid A : \mathcal{N})\\
&\phantom{\mathbf{data}\,\mathsf{F}_3() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_2 : (\vdash \mathsf{F}_3() \mid B : \mathcal{N})\\
\approx_{\sim L}\ &\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : ({\sim}A : \mathcal{V} \vdash \mathsf{F}_4() \mid)\\
&\phantom{\mathbf{data}\,\mathsf{F}_4() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_2 : (\vdash \mathsf{F}_4() \mid B : \mathcal{N})\\
\approx_{\sim L}\ &\mathbf{data}\,\mathsf{F}_5() : \mathcal{V}\,\mathbf{where}\,\mathsf{K}_1 : ({\sim}A : \mathcal{V} \vdash \mathsf{F}_5() \mid)\\
&\phantom{\mathbf{data}\,\mathsf{F}_5() : \mathcal{V}\,\mathbf{where}\,}\mathsf{K}_2 : ({\sim}B : \mathcal{V} \vdash \mathsf{F}_5() \mid)\\
\approx_{\oplus L}\ &\mathbf{data}\,\mathsf{F}_6() : \mathcal{V}\,\mathbf{where}\,\mathsf{K} : (({\sim}A) \oplus ({\sim}B) : \mathcal{V} \vdash \mathsf{F}_6() \mid)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{codata}\,\mathsf{G}_1() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_1() \vdash \neg(A \oplus B) : \mathcal{N})\\
\approx_{\neg R}\ &\mathbf{codata}\,\mathsf{G}_2() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (A \oplus B : \mathcal{V} \mid \mathsf{G}_2() \vdash)\\
\approx_{\oplus L}\ &\mathbf{codata}\,\mathsf{G}_3() : \mathcal{N}\,\mathbf{where}\begin{array}{l}\mathsf{O}_1 : (A : \mathcal{V} \mid \mathsf{G}_3() \vdash)\\ \mathsf{O}_2 : (B : \mathcal{V} \mid \mathsf{G}_3() \vdash)\end{array}\\
\approx_{\oplus L}\ &\mathbf{codata}\,\mathsf{G}_4() : \mathcal{N}\,\mathbf{where}\begin{array}{l}\mathsf{O}_1 : (\mid \mathsf{G}_4() \vdash \neg A : \mathcal{N})\\ \mathsf{O}_2 : (B : \mathcal{V} \mid \mathsf{G}_4() \vdash)\end{array}\\
\approx_{\oplus L}\ &\mathbf{codata}\,\mathsf{G}_5() : \mathcal{N}\,\mathbf{where}\begin{array}{l}\mathsf{O}_1 : (\mid \mathsf{G}_5() \vdash \neg A : \mathcal{N})\\ \mathsf{O}_2 : (\mid \mathsf{G}_5() \vdash \neg B : \mathcal{N})\end{array}\\
\approx_{\& R}\ &\mathbf{codata}\,\mathsf{G}_6() : \mathcal{N}\,\mathbf{where}\,\mathsf{O} : (\mid \mathsf{G}_6() \vdash (\neg A) \mathbin{\&} (\neg B) : \mathcal{N})
\end{aligned}$$

Shift laws

The last group of polarized connectives, the shifts, have not appeared in any of the algebraic or duality laws here. That is partially because their role is not to represent the structural aspects of (co-)data types—like the ability to contain several components or offer multiple alternatives—but instead serve to explicitly signal the mechanisms, like the ability to delay a computation and force it later, that integrate different evaluation strategies. In fact, the presence of shifts have the effect of *prohibiting* the usual algebraic and dual laws of polarized types which we see in practice in functional programming languages.

Returning to the examples of unfaithful encodings from the introduction, consider again the problem of encoding triples in terms of pairs in a call-by-name language like Haskell, where lazy pairs are described by the $\times_{\mathcal{N}}$ data type declared previously in Section 8.1, and lazy triples are represented as:

$$\begin{aligned}&\textbf{data}\ \mathsf{LazyTriple}(X{:}\mathcal{N}, Y{:}\mathcal{N}, Z{:}\mathcal{N}) : \mathcal{N}\ \textbf{where}\\ &\quad \mathsf{L3} : (X{:}\mathcal{N}, Y{:}\mathcal{N}, Z{:}\mathcal{N} \vdash \mathsf{LazyTriple}(X, Y, Z) \mid)\end{aligned}$$

By applying the polarization encoding from Figure 8.3 to a collection of declarations $\mathcal{G}$ containing both $\times_{\mathcal{N}}$ and $\mathsf{LazyTriple}$, we get that[6]

$$X : \mathcal{N}, Y : \mathcal{N}, Z : \mathcal{N} \vDash [\![\mathsf{LazyTriple}(X, Y, Z)]\!]_{\mathcal{G}} \approx \Uparrow(\downarrow X \otimes (\downarrow Y \otimes \downarrow Z))$$
$$X : \mathcal{N}, Y : \mathcal{N}, Z : \mathcal{N} \vDash [\![X \times_{\mathcal{N}} (Y \times_{\mathcal{N}} Z)]\!]_{\mathcal{G}} \approx \Uparrow(\downarrow X \otimes \downarrow\Uparrow(\downarrow Y \otimes \downarrow Z))$$

but these two types represent very different space of possible program behaviors because of the extra shifts in the encoding of $X \times_{\mathcal{N}} (Y \times_{\mathcal{N}} Z)$. In other words, the difference between the two is that the type $X \times_{\mathcal{N}} (Y \times_{\mathcal{N}} Z)$ allows for extra values like $\mathsf{Pair}_{\mathcal{N}}(x, \mu\_. \langle y \| \beta \rangle)$, where $\mu\_. \langle y \| \beta \rangle$ is a term that does not return any result, but $\mathsf{LazyTriple}(X, Y, Z)$ does not, which is explicitly expressed by the presence or absence of shifts in their encoding. Furthermore, whereas we can apply properties like associativity of $\otimes$ within the encoding of $\mathsf{LazyTriple}(X, Y, Z)$, where

$$X : \mathcal{N}, Y : \mathcal{N}, Z : \mathcal{N} \vDash \Uparrow(\downarrow X \otimes (\downarrow Y \otimes \downarrow Z)) \approx \Uparrow((\downarrow X \otimes \downarrow Y) \otimes \downarrow Z)$$

this is blocked by the extra shifts in $\Uparrow(\downarrow X \otimes \downarrow\Uparrow(\downarrow Y \otimes \downarrow Z))$, which prevent the law from applying.

We can also view the troubles with currying in a call-by-value language like ML in terms of extra shifts by with the representation of call-by-value functions as the $\to_{\mathcal{V}}$ co-data type previously declared in Section 8.1, whose encoding simplifies to

$$X : \mathcal{V}, Y : \mathcal{V} \vDash [\![X \to_{\mathcal{V}} Y]\!]_{\mathcal{G}} \approx \Downarrow(\neg X \parr \uparrow Y)$$

[6] More specifically, the immediate output of translation is $[\![\mathsf{LazyTriple}(X, Y, Z)]\!]_{\mathcal{G}} \triangleq \Uparrow((\downarrow X \otimes (\downarrow Y \otimes (\downarrow Z \otimes 1))) \oplus 0)$ and $[\![X \times_{\mathcal{N}} Y]\!]_{\mathcal{G}} \triangleq \Uparrow((\downarrow X \otimes (\downarrow Y \otimes 1)) \oplus 0)$, which is cleaned up as shown by the laws in Section 8.4.

$$\downarrow_{\mathcal{V}} A \approx A \approx {}_{\mathcal{V}}{\Uparrow} A \qquad\qquad \uparrow_{\mathcal{N}} A \approx A \approx {}_{\mathcal{N}}{\Downarrow} A$$

FIGURE 8.10. Identity laws of the redundant self-shift connectives.

Again, the shifts get in the way when we try to apply the algebraic or logical laws of the polarized connectives. The type of uncurried call-by-value functions is

$$X : \mathcal{V}, Y : \mathcal{V}, Z : \mathcal{V} \vDash \llbracket (X \otimes Y) \to_{\mathcal{V}} Z \rrbracket \approx {\Downarrow}(\neg(X \otimes Y) \parr {\uparrow} Z) \approx {\Downarrow}(\neg X \parr \neg Y \parr {\uparrow} Z)$$

whereas the type of curried call-by-value functions is

$$X : \mathcal{V}, Y : \mathcal{V}, Z : \mathcal{V} \vDash \llbracket X \to_{\mathcal{V}} (Y \to_{\mathcal{V}} Z) \rrbracket \approx {\Downarrow}(\neg X \parr {\uparrow}{\Downarrow}(\neg Y \parr {\uparrow} Z))$$

which is not the same because of the extra shifts.

This does not mean that the shifts are completely lawless, however. Since we began with a large family of shifts—singleton data and co-data type constructors mapping between any kind $\mathcal{S}$ and $\mathcal{V}$ or $\mathcal{N}$—some of them turn out to be redundant as shown in Figure 8.10. The data shifts $\downarrow_{\mathcal{V}}$ and ${}_{\mathcal{V}}{\Uparrow}$ for wrapping call-by-value type as another call-by-value type and the co-data shifts $\uparrow_{\mathcal{N}}$ and ${}_{\mathcal{N}}{\Downarrow}$ for doing the same to call-by-name types are all identity operations on types, up to isomorphism. In particular, the data declarations for $\downarrow_{\mathcal{V}}$ and ${}_{\mathcal{V}}{\Uparrow}$ are the simplest instance of Lemma 8.5 (a) which means that $\downarrow_{\mathcal{V}} A \approx A \approx {}_{\mathcal{V}}{\Uparrow} A$, and likewise $\uparrow_{\mathcal{N}} A \approx A \approx {}_{\mathcal{N}}{\Downarrow} A$ because of Lemma 8.5 (b). This fact tells us that the polarizing translation on already-polarized types is actually an identity up to isomorphism, i.e. for any $\Theta \vdash_{\mathcal{P}} A : \mathcal{S}$, it follows that $\Theta \vDash \llbracket A \rrbracket_{\mathcal{P}} \approx A$. For example, we have

$$X : \mathcal{V}, Y : \mathcal{V} \vDash \llbracket X \oplus Y \rrbracket_{\mathcal{P}} \triangleq {}_{\mathcal{V}}{\Uparrow}(((\downarrow_{\mathcal{V}} X \otimes 1) \oplus (\downarrow_{\mathcal{V}} Y \otimes 1)) \oplus 0) \approx X \oplus Y$$

for the additive data type, and

$$X : \mathcal{V} \vDash \llbracket \neg X \rrbracket_{\mathcal{P}} \triangleq {}_{\mathcal{N}}{\Downarrow}(\top \mathbin{\&} (\bot \parr (\neg \downarrow_{\mathcal{V}} X))) \approx \neg X$$

for the negation co-data type, justifying our rule of thumb for deciding the appropriate strategies for the polarized basis $\mathcal{P}$ of (co-)data types.

Functional laws

So far, our attention has been largely focused on properties of the polarized (co-)data types from Figure 8.1, some of which, like $\parr$, are unfamiliar as programming constructs. But what about a more familiar construct like functions? We have seen that call-by-value functions don't behave as nicely as we'd like, which can be understood as unfortunate extra shifts between evaluation strategies. So is there a type of function that avoids these problems? As it turns out, the mixed-polarity, "primordial" (Zeilberger, 2009) function type that we considered in Chapter IV captures the best of both the call-by-value and call-by-name worlds, which is represented by the co-data declaration:

$$\mathbf{codata}\,(X{:}\mathcal{V} \to Y{:}\mathcal{N}) : \mathcal{N}\ \mathbf{where}$$

$$\_ \cdot \_ : (X : \mathcal{V} \mid X \to Y \vdash Y : \mathcal{N})$$

The particular placement of $\mathcal{V}$ and $\mathcal{N}$ again follows the rule of thumb from Section 8.1, so as a consequence the polarized encoding for $A \to B$ avoids any impactful shifts. Because of the identity laws for shifts from Figure 8.10, the polarizing encoding for the above declaration $\mathcal{G}$ simplifies down to just $\neg$ and $\parr$:

$$X : \mathcal{V}, Y : \mathcal{N} \vDash [\![X \to Y]\!]_{\mathcal{G}} \approx {}_{\mathcal{N}}\Downarrow(\neg(\downarrow_{\mathcal{V}} X) \parr (\uparrow_{\mathcal{N}} Y)) \approx \neg X \parr Y$$

This gives us the most primitive expression of functions in our multi-strategy language; the rest can be encoded in terms of the above polarized function type by adding back the extra shifts.

Alternatively, we could have chosen to replace the unfamiliar $\parr$ with this function type. Because of the involutive nature of the $\neg$ and $\sim$ negations, we have the following encoding of $\parr$ in terms of $\to$ and $\sim$: $A \parr B \approx \neg(\sim A) \parr B \approx (\sim A) \to B$. Certainly functions are more familiar than $\parr$ as a programming construct, but the cost of leaning on this familiarity is the loss of symmetry because functions are a "half-negated or." In particular, we can recast all of the algebraic and logical laws about $\parr$ in terms of $\to$ as shown in Figure 8.11, some of which are familiar properties of implication, that are all derived from the encoding $A \to B \approx \neg A \parr B$. The commutativity, associativity, and unit laws of the underlying $\parr$ give us contrapositive, currying, thunking, and negating laws:

$$A \to B \approx (\neg A) \parr B \approx B \parr (\neg A) \approx (\neg(\sim B)) \parr (\neg A) \approx (\sim B) \to (\neg A)$$

$$
\begin{aligned}
A \to B &\approx ({\sim}B) \to (\neg A) & A \to (B \mathbin{\&} C) &\approx (A \to B) \mathbin{\&} (A \to C) \\
(A \otimes B) \to C &\approx A \to (B \to C) & (A \oplus B) \to C &\approx (A \to C) \mathbin{\&} (B \to C) \\
1 \to A &\approx A & A \to \top &\approx \top \\
A \to \bot &\approx \neg A & 0 \to A &\approx \top
\end{aligned}
$$

$$
\begin{aligned}
{\sim}(A \to B) &\approx A \otimes ({\sim}B) \\
A \to (\neg B) &\approx \neg(A \otimes B)
\end{aligned}
$$

FIGURE 8.11. Derived laws of polarized functions.

$$
\begin{aligned}
(A \otimes B) \to C &\approx (\neg(A \otimes B)) \parr C \approx ((\neg A) \parr (\neg B)) \parr C \\
&\approx (\neg A) \parr ((\neg B) \parr C) \approx A \to (B \to C) \\
1 \to A &\approx (\neg 1) \parr A \approx \bot \parr A \approx A \\
A \to \bot &\approx (\neg A) \parr \bot \approx \neg A
\end{aligned}
$$

Likewise, distributing $\parr$ over & and annihilating it with $\top$ recognize certain functions as products or trivial units:

$$
\begin{aligned}
A \to (B \mathbin{\&} C) &\approx (\neg A) \parr (B \mathbin{\&} C) \approx ((\neg A) \parr B) \mathbin{\&} ((\neg A) \parr C) \\
&\approx (A \to B) \mathbin{\&} (A \to C) \\
(A \oplus B) \to C &\approx (\neg(A \oplus B)) \parr C \approx ((\neg A) \mathbin{\&} (\neg B)) \parr C \\
&\approx ((\neg A) \parr C) \mathbin{\&} ((\neg B) \parr C) \approx (A \to C) \mathbin{\&} (B \to C) \\
A \to \top &\approx (\neg A) \parr \top \approx \top \\
0 \to A &\approx (\neg 0) \parr A \approx \top \parr A \approx \top
\end{aligned}
$$

And finally, the De Morgan duality between $\parr$ and $\otimes$ tells us that the continuation of a function is a pair, and that a continuation for a pair is a function:

$$
\begin{aligned}
{\sim}(A \to B) &\approx {\sim}((\neg A) \parr B) \approx ({\sim}(\neg A)) \otimes ({\sim}B) \approx A \otimes ({\sim}B) \\
A \to (\neg B) &\approx (\neg A) \parr (\neg B) \approx \neg(A \otimes B)
\end{aligned}
$$

The Faithfulness of Polarization

Now that we have laid down some laws for declaration isomorphisms, we can put them to use for encoding user-defined (co-)data types in terms of the polarized connectives from Figure 8.1. In particular, we can extend the laws from Figures 8.6 and 8.7 for polarized sub-structures appearing within a simple singleton declaration to apply to any general (co-)data type using the mix laws from Figures 8.4 and 8.5. For example, given a declaration of the form

$$\begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \quad \mathsf{K}_0 : (\Gamma_0, A : \mathcal{V}, B : \mathcal{V} \vdash \mathsf{F}(\Theta) \mid \Delta_0) \\ \quad \overrightarrow{\mathsf{K} : (\Gamma \vdash \mathsf{F}(\Theta) \mid \Delta)} \end{array}$$

we can combine the A and B components of the K_0 constructor with the $\otimes$ connective by starting with the $\otimes L$ law, and then building up to the full declaration of F by applying the mix law to the appropriate reflexive isomorphisms as discussed in Section 8.3 as follows:

$$\begin{array}{c} \begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \quad \mathsf{K} : (A : \mathcal{V}, B : \mathcal{V} \vdash \mathsf{F}(\Theta) \mid\,) \end{array} \approx \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \quad \mathsf{K}' : (A \otimes B : \mathcal{V} \vdash \mathsf{F}'(\Theta) \mid\,) \end{array} \\ \hline \begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \quad \mathsf{K}_0 : (A : \mathcal{V}, B : \mathcal{V}, \Gamma_0 \vdash \mathsf{F}(\Theta) \mid \Delta_0) \end{array} \approx \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \quad \mathsf{K}'_0 : (A \otimes B : \mathcal{V}, \Gamma_0 \vdash \mathsf{F}'(\Theta) \mid \Delta_0) \end{array} \\ \hline \begin{array}{l} \textbf{data}\,\mathsf{F}(\Theta) : \mathcal{V}\,\textbf{where} \\ \quad \mathsf{K}_0 : (A : \mathcal{V}, B : \mathcal{V}, \Gamma_0 \vdash \mathsf{F}(\Theta) \mid \Delta_0) \\ \quad \overrightarrow{\mathsf{K} : (\Gamma \vdash \mathsf{F}(\Theta) \mid \Delta)} \end{array} \approx \begin{array}{l} \textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{V}\,\textbf{where} \\ \quad \mathsf{K}'_0 : (A \otimes B : \mathcal{V}, \Gamma_0 \vdash \mathsf{F}'(\Theta) \mid \Delta_0) \\ \quad \overrightarrow{\mathsf{K} : (\Gamma \vdash \mathsf{F}'(\Theta) \mid \Delta)} \end{array} \end{array}$$

Other combinations of components at different positions in constructors of F can be targeting with the commute laws for data declarations. This idea is the central technique of the encoding, which just repeats the above procedure until we are left with only a singleton (co-)data type that "wraps" its encoding. First we consider how to encode a just one (co-)data type declaration in terms of polarized connectives.

Theorem 8.9 (Polarizing (co-)data declarations)**.** *For any $\mathcal{S}$ validating $\chi_{\mathcal{S}}$,*

a) given $\textbf{data}\,\mathsf{F}(\Theta) : \mathcal{S}\,\textbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash \mathsf{F}(\Theta) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$*, we have* $\Theta \vDash \mathsf{F}(\Theta) \approx [\![\mathsf{F}(\Theta)]\!]_{\mathcal{G}}$*, and* *b) given* **codata** $\mathsf{G}(\Theta) : \mathcal{S}$ **where** $\overrightarrow{\mathsf{O}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\Theta) \vdash \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \in \mathcal{G}$*, we have* $\Theta \vDash \mathsf{G}(\Theta) \approx [\![\mathsf{G}(\Theta)]\!]_{\mathcal{G}}$.

Proof. a) Observe that we have the following data isomorphism by extending the polarized laws from Figure 8.6 with the mix and commute laws from Figure 8.4:

$$
\begin{aligned}
&\textbf{data}\,\mathsf{F}_1(\Theta) : \mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash \mathsf{F}_1(\Theta) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \\
\approx_{\downarrow L}\ &\textbf{data}\,\mathsf{F}_2(\Theta) : \mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij} : \mathcal{V}}^{j} \vdash \mathsf{F}_2(\Theta) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \\
\approx_{\uparrow R}\ &\textbf{data}\,\mathsf{F}_3(\Theta) : \mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij} : \mathcal{V}}^{j} \vdash \mathsf{F}_3(\Theta) \mid \overrightarrow{\uparrow_{\mathcal{R}_{ij}} B_{ij} : \mathcal{N}}^{j}\right)}^{i} \\
\approx_{\sim L}\ &\textbf{data}\,\mathsf{F}_4(\Theta) : \mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij} : \mathcal{V}}^{j}, \overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}} B_{ij}) : \mathcal{V}}^{j} \vdash \mathsf{F}_4(\Theta) \mid\ \right)}^{i} \\
\approx_{1L,\otimes L}\ &\textbf{data}\,\mathsf{F}_5(\Theta) : \mathcal{V}\,\textbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\bigotimes\left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij}}^{j}, \overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}} B_{ij})}^{j}\right) : \mathcal{V} \vdash \mathsf{F}_5(\Theta) \mid\ \right)}^{i} \\
\approx_{0L,\oplus L}\ &\begin{aligned}&\textbf{data}\,\mathsf{F}_6(\Theta) : \mathcal{V}\,\textbf{where} \\ &\quad \mathsf{K} : \left(\bigoplus\left(\overrightarrow{\bigotimes\left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij}}^{j}, \overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}} B_{ij})}^{j}\right)}^{i}\right) : \mathcal{V} \vdash \mathsf{F}_6(\Theta) \mid\ \right)\end{aligned}
\end{aligned}
$$

With the above isomorphism between F_1 and F_6, it follows from the data shift law that:

$$
\begin{aligned}
&\textbf{data}\,\mathsf{F}(\Theta) : \mathcal{S}\,\textbf{where}\,\overrightarrow{\mathsf{K}_i : \left(\overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash \mathsf{F}(\Theta) \mid \overrightarrow{B_{ij} : \mathcal{R}_{ij}}^{j}\right)}^{i} \\
\approx\ &\textbf{data}\,\mathsf{F}'(\Theta) : \mathcal{S}\,\textbf{where}\,\mathsf{K} : \left(\bigoplus\left(\overrightarrow{\bigotimes\left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij}}^{j}, \overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}} B_{ij})}^{j}\right)}^{i}\right) : \mathcal{V} \vdash \mathsf{F}'(\Theta) \mid\right)
\end{aligned}
$$

We then get that

$$
\Theta \vDash \mathsf{F}'(\Theta) \approx {}_{\mathcal{S}}\Uparrow\left(\bigoplus\left(\overrightarrow{\bigotimes\left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij}}^{j}, \overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}} B_{ij})}^{j}\right)}^{i}\right)\right) \triangleq [\![\mathsf{F}(\Theta)]\!]_{\mathcal{G}}
$$

by applying Lemma 8.3 (a) to the reflexive isomorphism of

$$
\left(\bigoplus\left(\overrightarrow{\bigotimes\left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij}}^{j}, \overrightarrow{\sim(\uparrow_{\mathcal{R}_{ij}} B_{ij})}^{j}\right)}^{i}\right)\right)
$$

so by transitivity $\Theta \vDash \mathsf{F}(\Theta) \approx [\![\mathsf{F}(\Theta)]\!]_{\mathcal{G}}$.

b) Analogous to the proof of Theorem 8.9 (a) by duality. □

Now that we know how to encode individual (co-)data types in isolation, we look to a global encoding of arbitrary types made out of a collection $\mathcal{G}$ of (co-)data declarations. The only limitation on the group of declarations $\mathcal{G}$ is that they be well-formed and non-cyclic, which is a consequence of the judgement $\left(\vdash_{\mathcal{G}}\right)\,\mathbf{seq}$ from Section 6.2. The non-cyclic requirement ensures that the dependency chains between declarations is well-founded, so the process of inlining the encodings of (co-)data types will eventually terminate and give a final, fully-expanded encoding.

Theorem 8.10 ((Co-)Data Polarization)**.** *Given derivations of both* $\left(\vdash_{\mathcal{G}}\right)\,\mathbf{seq}$ *and* $\Theta \vdash_{\mathcal{G}} A : \mathcal{S}$*, it follows that* $\Theta \vDash A \approx [\![A]\!]_{\mathcal{G}}$*.*

Proof. By lexicographic induction on (1) the derivation of $\left(\vdash_{\mathcal{G}}\right)\,\mathbf{seq}$, and (2) the derivation of $\Theta \vdash_{\mathcal{G}} A : \mathcal{S}$. The case when A is a variable is immediate. The case where $A = \mathsf{F}(\vec{C})$ for some F declared in $\mathcal{G}$ as

$$\mathbf{data}\,\mathsf{F}(\overrightarrow{X : \mathcal{S}'}) : \mathcal{S}\,\mathbf{where}\,\overrightarrow{\mathsf{K} : \left(\overrightarrow{A : \mathcal{T}} \vdash \mathsf{F}(\vec{X}) \mid \overrightarrow{B : \mathcal{R}}\right)}$$

follows from Theorems 8.9 and 8.8. In particular, we have

$$\overrightarrow{\Theta \vDash C \approx [\![C]\!]_{\mathcal{G}}} \qquad \overrightarrow{X : \mathcal{S}'} \vDash A_{ij} \approx [\![A_{ij}]\!]_{\mathcal{G}'} \qquad \overrightarrow{X : \mathcal{S}'} \vDash B_{ij} \approx [\![B_{ij}]\!]_{\mathcal{G}'}$$

from the inductive hypothesis for some $\mathcal{G}'$ strictly smaller than $\mathcal{G}$. From Theorem 8.9, we have

$$\mathsf{F}(\vec{C}) \approx {}_{\mathcal{S}}\Uparrow(\bigoplus(\overrightarrow{\bigotimes(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij}\theta}^{j}, {\sim}(\overrightarrow{\uparrow_{\mathcal{R}_{ij}} B_{ij}\theta}^{j}))}^{i}))$$

where $\theta = \overrightarrow{\{C/X\}}$, and from Theorem 8.8 we know that

$$\begin{aligned}\Theta \vDash {}_{\mathcal{S}}\Uparrow &\left(\bigoplus\left(\overrightarrow{\bigotimes\left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} A_{ij}\theta}^{j}, {\sim}(\overrightarrow{\uparrow_{\mathcal{R}_{ij}} B_{ij}\theta}^{j})\right)}^{i}\right)\right) \\ \approx {}_{\mathcal{S}}\Uparrow &\left(\bigoplus\left(\overrightarrow{\bigotimes\left(\overrightarrow{\downarrow_{\mathcal{T}_{ij}} [\![A_{ij}]\!]_{\mathcal{G}}[\![\theta]\!]_{\mathcal{G}}}^{j}, \overrightarrow{{\sim}(\uparrow_{\mathcal{R}_{ij}} [\![B_{ij}]\!]_{\mathcal{G}}[\![\theta]\!]_{\mathcal{G}})}^{j}\right)}^{i}\right)\right)\end{aligned}$$

where $[\![\theta]\!]_{\mathcal{G}} = \overrightarrow{\left\{[\![C]\!]_{\mathcal{G}}/X\right\}}$. Therefore, we have $\Theta \vDash \mathsf{F}(\vec{C}) \approx \left[\!\!\left[\mathsf{F}(\vec{C})\right]\!\!\right]_{\mathcal{G}}$ by distributing the substitution θ over translation. The case where $A = \mathsf{G}(\vec{C})$ for some G declared in $\mathcal{G}$ as co-data follows similarly. $\square$

Note that as an immediate consequence of the full (co-)data polarization encoding (Theorem 8.10), we can generalize the fact that isomorphism distributes over substitution into a type made from polarized connectives (Theorem 8.8) to conclude that isomorphism distributes over substitution into *any* type built from (non-cyclic) (co-)data type constructors. In particular, for any non-cyclically well-formed $\mathcal{G}$, $\Theta, X : \mathcal{S} \vdash_{\mathcal{G}} A : \mathcal{T}$, $\Theta \vdash_{\mathcal{G}} B : \mathcal{S}$, and $\Theta \vdash_{\mathcal{G}} C : \mathcal{S}$, if $\Theta \vDash B \approx C$ then $\Theta \vDash A\{B/X\} \approx A\{C/X\}$. This fact means that we can apply any isomorphism within the context of any encodable (co-)data type.

CHAPTER IX

Representing Functional Programs

At this point, we have looked at the design and theory of many different programming language features in the setting of the sequent calculus. We looked at a symmetric mechanism for user-defined data and co-data types, an abstract treatment of evaluation strategy in terms of substitution that lets us mix multiple evaluation orders within a program (Chapter V). We also looked at ways to do capture type abstraction as higher-order data and co-data types, and well-founded recursion in both types and programs (Chapter VI). But how has what we learned impact functional programming languages which are based on natural deduction and the λ-calculus? Is there some way to transfer ideas born in the sequent calculus over to more traditional programming languages?

Yes! The sequent calculus was developed as a tool for studying natural deduction (Gentzen, 1935a), so the two already have a well-established relationship (Gentzen, 1935a). Here, we will now employ that canonical relationship between the two logics to develop the natural deduction, λ-based counterpart to everything we have done in the $\mu\tilde{\mu}$ sequent calculus. This lets us talk about ideas like user-defined (co-)data types, multiple kinds of evaluation strategies, and mixed induction and co-induction in a core language that is much closer to pure functional programming.

The pure λ-calculus counterpart is not just loosely based on the ideas developed in the $\mu\tilde{\mu}$-calculus, the two are in a close correspondence between their static and dynamic semantics. In particular, by limiting ourselves to just one consequence, which eliminates the possibility of control effects, typability and equations between program fragments in the two languages are in a one-for-one correspondence. The correspondence between sequent calculus and natural deduction has two applications. First, it lets us compile functional programs, which come from languages based on the λ-calculus, so a sequent calculus representation. Second, it lets us transfer results, such as strong normalization (in Chapter VII) found in the sequent-based language to the natural deduction one. So the single-consequence $\mu\tilde{\mu}$-calculus can be seen as a more machine-like version of a pure λ-calculus core language which serves as a compile target as well as a good vehicle for studying the properties of functional programs.

Next, we consider what natural deduction language corresponds to the entire, multiple-consequence, $\mu\tilde{\mu}$-calculus. Multiple consequences are neatly accommodated by Parigot's (1992) $\lambda\mu$-calculus, which is a form of natural deduction for classical logic, and a term language for first-class control effects. This extension sacrifices purity for greater expressivity (Felleisen, 1991) which has practical applications in compilers. In particular, optimizing compilers rely on the idea of *join points*—a representation of shared control flow in a program which joins back together after diverging across different branching paths, like after an if-then-else construct—for preventing code size explosion while transforming programs.

In common compiler intermediate languages, join points are represented as ϕ-nodes in static single assignment (SSA) (Cytron *et al.*, 1991) and as just continuations in continuation-passing style (CPS) (Appel, 1992). However, the proper treatment of join points is typically a troublesome issue in languages based on a pure direct-style λ-calculus (Kennedy, 2007). In contrast, the direct-style $\lambda\mu$-calculus presented here can be used as a basis for a compiler intermediate language for functional languages that allows for the proper expression of shared control flow in terms of general first-class control. If we are interested in compiling pure functional programs in particular, we can restrict this calculus down to the pure subset without loosing the correct treatment of join points by limiting the types and occurrences of co-variables (Maurer *et al.*, 2017; Downen *et al.*, 2016). This restriction also makes it more direct to give a good account of recursive join points which are helpful for implementing efficient loops for functional languages.

This chapter covers the following topics:

- A λ-calculus based natural deduction counterpart to the $\mu\tilde{\mu}$ sequent calculus, λlet (Section 9.1), including all of the language features we have considered so far: user-defined, higher-order (co-)data types, mixed evaluation strategies, and well-founded recursion.

- The direct correspondence between the static and dynamic semantics of λlet and the single-consequence restriction $\mu\tilde{\mu}$ (Section 9.2), which is "direct" in the sense that the translation between the two languages is local, not global, so that the types of terms are exactly the same in both languages (unlike continuation-passing style transformations).

- The multipe-consequence extension of the pure natural deduction λlet-calculus, $\lambda\mu let$ (Section 9.3), which heightens the correspondence to cover the full $\mu\tilde{\mu}$ sequent calculus and allows for the direct representation of shared control flow.

Pure Data and Co-Data in Natural Deduction

So far, we have looked at a calculus in sequent style, which corresponds to a classical logic and thus includes control effects (Griffin, 1990). Let's now shift focus, and see how the intuition we gained from the sequent calculus can be reflected back into a more traditional core calculus for representing functional programs. The goal here is to see how the principles we have developed in the sequent setting can be incorporated into a λ-calculus based language: using the traditional connection between natural deduction and the sequent calculus, we show how to translate our primitive and noetherian recursive types and programs into natural deduction style. In essence, we will consider a functional calculus based on an effect-free subset of the $\mu\tilde{\mu}$-calculus corresponding to Gentzen's (1935a) LJ sequent calculus for intuitionistic logic.

Static semantics

Essentially, the intuitionistic restriction of the $\mu\tilde{\mu}$ sequent calculus for representing effect-free programs follows a single mantra, based on the connection between the classical and intuitionistic logics LK and LJ: there is always *exactly* one conclusion. In the type system, this means that the sequent for typing terms has the more restricted form $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A$, where the active type on the right is no longer ambiguous and does not need to be distinguished (with $|$), as is more traditional for functional languages. Notice that this limitation on the form of sequents impacts which data and co-data types we can express. For example, the common sums and products, which were declared as

$$\begin{array}{lll} \textbf{data}\, X \oplus Y\, \textbf{where} & \qquad & \textbf{codata}\, X \,\&\, Y\, \textbf{where} \\ \quad \iota_1 : \; X \vdash X \oplus Y \mid & & \quad \pi_1 : \; \mid X \,\&\, Y \vdash X \\ \quad \iota_2 : \; X \vdash X \oplus Y \mid & & \quad \pi_2 : \; \mid X \,\&\, Y \vdash Y \end{array}$$

fit into this restricted typing discipline, because each of their constructors and observers involves exactly one type to the right of entailment. However, the (co-)data types for representing more exotic connectives like the two negations

$$\begin{array}{ll}\textbf{data}\,{\sim}X\,\textbf{where} & \textbf{codata}\,\neg Y\,\textbf{where} \\ \quad {\sim} : \quad \vdash {\sim}X \mid X & \quad \neg : \quad X \mid \neg Y \vdash\end{array}$$

or the binary and nullary disjunctive co-data types

$$\begin{array}{ll}\textbf{codata}\,X \parr Y\,\textbf{where} & \textbf{codata}\,\bot\,\textbf{where} \\ \quad [\_,\_] : \quad \mid X \parr Y \vdash X, Y & \quad [] : \quad \mid \bot \vdash\end{array}$$

do not fit, because they require placing zero or two types to the right of entailment. In sequent style, this means these *pure* data types can never contain a co-value, and *pure* co-data types must always involve exactly one co-value for returning the unique result. In functional style, the data types are exactly the algebraic data types used in functional languages, with the corresponding constructors and case expressions, and the co-data types can be thought of as merging functions with records into a notion of abstract "objects" which compute and return a value when observed. For example, to observe a value of type $X \,\&\, Y$, we could access the first component as a record field, $v\ \pi_1$, and we describe an object of this type by saying how it responds to all possible observations, $\lambda\{\pi_1 \Rightarrow v_1 \mid \pi_2 \Rightarrow v_2\}$, with the typing rules:

$$\frac{\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : B}{\Gamma \vdash \lambda\{\pi_1 \Rightarrow v_1 \mid \pi_2 \Rightarrow v_2\} : A \,\&\, B} \qquad \frac{\Gamma \vdash v : A \,\&\, B}{\Gamma \vdash v\ \pi_1 : A} \quad \frac{\Gamma \vdash v : A \,\&\, B}{\Gamma \vdash v\ \pi_2 : B}$$

Likewise, the traditional λ-abstractions and type abstractions from system F (as seen previously in Chapter II) can be expressed by objects of these form. Specifically, since they definable (as seen previously in Sections 5.2 and 6.2) as pure co-data types with one observer, $\_ \cdot \_ : (X \mid X \to Y \vdash Y)$ and $\_ \,@\, \_ : \left(\mid \forall(X) \vdash^{Y:k} X\ Y \right)$ respectively, so that the application of a function v to an argument v' is written as $v \cdot v'$, the specification of the polymorphic v to a type A is written as $v \mathbin{@} A$, and the basic λ-abstractions are syntactic sugar that removes the extra generality:

$$\lambda x.v \triangleq \lambda\{\,\cdot x \Rightarrow v\} \qquad\qquad \Lambda Y{:}k.v \triangleq \lambda\{\,@\,Y{:}k \Rightarrow v\}$$

Thus, these objects serve as "generalized λ-abstractions" (Abel & Pientka, 2013) defined by shallow case analysis rather than deep pattern-matching.

$$
\begin{array}{rll}
v \in \mathit{Term} ::= & x \mid \mathbf{let}\, x = v \,\mathbf{in}\, v' & (\mathit{core}) \\
& \mid \mathsf{K}(\vec{B}, \vec{v}) \mid \mathbf{case}\, v' \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} & (\mathit{data}) \\
& \mid \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\right\} \mid v'\ O[\vec{B}, \vec{v}] & (\mathit{co\text{-}data}) \\
\\
F \in \mathit{FrameCxt} ::= & \square \mid \mathbf{let}\, x = F \,\mathbf{in}\, v & (\mathit{core\ frames}) \\
& \mid F\ \ \mathsf{O}[\vec{B}, \vec{v}] & (\mathit{co\text{-}data\ frames}) \\
& \mid \mathbf{case}\, F \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} & (\mathit{data\ frames})
\end{array}
$$

FIGURE 9.1. Untyped syntax for a natural deduction language of data and co-data.

Putting this more formally, the untyped syntax of $\lambda \mathit{let}$, a natural deduction style pure λ-calculus, is given in Figure 9.1. At its core, the $\lambda \mathit{let}$-calculus includes variables and **let** expressions, which allow for the binding and reference of names without imposing any particular structure. In addition, the untyped syntax of $\lambda \mathit{let}$ includes arbitrary data structures and case analysis on data structures (of the form $\mathsf{K}(\vec{B}, \vec{v})$ and $\mathbf{case}\, v' \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}$) as well as arbitrary co-data objects and observations of those objects (of the form $\lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\right\}$ and $v'\ O[\vec{B}, \vec{v}]$). The observations of the results of terms are given by the syntax of *frame contexts* F. While these contexts are a meta-syntactic construct (that is, they are contexts of the syntax of terms, but not syntax themselves), they will soon play a crucial in the dynamic semantics of $\lambda \mathit{let}$ to come.

On top of the untyped syntax, we have the static typing rules. The rules for the type-level upward and well-formed sequents are exactly the same as in the $\mu\tilde{\mu}$-calculus, so we do not repeat them here, and instead only present the typing rules for terms. First, there are the core typing rules in Figure 9.2 which correspond to the core of the $\mu\tilde{\mu}$-calculus: the *Var* rule corresponds to right-variable rule *VR*, the *Let* rule corresponds to the *Cut* rule, and the *TC* rule corresponds to the right-type conversion rule *TCR*. Note that weakening and contraction are built into these rules, following the style of natural deduction which makes structural inferences implicit. Next, we have the typing rules for pure data and co-data types in the $\lambda \mathit{let}$-calculus: the rules for simple multi-kinded (co-)data types are shown in Figure 9.3 and the more advanced rules for higher-order (co-)data types are shown in Figure 9.4. Intuitively, these rules

$$Judgement ::= \Gamma \vdash^{\Theta}_{\mathcal{G}} v : A$$

$$\frac{}{\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} x : A}\ Var \qquad \frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \quad \Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} v' : C}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{let}\, x = v\, \mathbf{in}\, v' : C}\ Let$$

$$\frac{\Theta \vdash_{\mathcal{G}} A =_{\beta\eta} B : k \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v : A}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : B}\ TC$$

FIGURE 9.2. A natural deduction language for the core calculus.

generalize the typing rules from the λ-calculus in Chapter II. Also note that for a **case** expression which introduces type variables in its branches, the associated elimination rule implicitly imposes the restriction that the return type cannot reference those type variables. The implicit restriction comes from the fact that, for the conclusion of the elimination rule to be well formed. That is, if we know that the sequent corresponding to $\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l}\,\vec{x}) \Rightarrow v_i}^{i} : C$ is well formed, i.e. if we have a derivation of $\left(\Gamma \vdash^{\Theta}_{\mathcal{G}} C\right) \mathbf{seq}$, then that implies that none of $\vec{Y}$ are free in C, since they are not already in Θ because we are able to extend the typing environment to $\Theta, \overrightarrow{Y : l}$ in the premise.

Dynamic semantics

With the static semantics for how natural deduction programs are formed, we now consider the dynamic semantics for how programs behave. As with the $\mu\tilde{\mu}$-sequent calculus, we will characterize the impact of evaluation strategy on substitution as a parameter to the language. In λlet, the corresponding notion of a *substitution strategy* $\mathcal{T}$ is a subset of terms called *values* ($V \in Value_{\mathcal{T}}$) and a subset of frame contexts called *co-values* ($E \in CoValue_{\mathcal{T}}$), such that variables are values, the empty context is a co-value, and co-values compose (i.e. if E and E' are co-values then so is $E[E']$). Next, an *evaluation strategy* $\mathcal{T}$ includes a substitution strategy as well as a subset of all contexts called *evaluation contexts* ($D \in EvalCxt_{\mathcal{T}}$) such that every co-value is an evaluation context. Note that the scope of potential evaluation contexts is quite large, and co-values point out a special subset of all evaluation contexts. In essence, Given **data** $\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}$ **where** $\overrightarrow{\mathsf{K}_i : \overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash \mathsf{F}(\overrightarrow{X})}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A_{ij} \left\{\overrightarrow{B/X}\right\}}^{j}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{K}_i(\overrightarrow{v}) : \mathsf{F}(\overrightarrow{B})}\ \mathsf{F}I_{\mathsf{K}_i}$$

$$\frac{\Theta \vdash_{\mathcal{G}} \mathsf{F}(\overrightarrow{B}) : \mathcal{S} \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v : \mathsf{F}(\overrightarrow{B}) \quad \overrightarrow{\Gamma, \overrightarrow{x : A_{ij} \left\{\overrightarrow{B/X}\right\}}^{j} \vdash^{\Theta}_{\mathcal{G}} v_i : C}^{i}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}_i(\overrightarrow{x}) \Rightarrow v_i}^{i} : C}\ \mathsf{F}E$$

Given **codata** $\mathsf{G}(\overrightarrow{X{:}k}) : \mathcal{S}$ **where** $\overrightarrow{\mathsf{O}_i : \overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\overrightarrow{X}) \vdash A'_i : \mathcal{R}_i}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{\Gamma, \overrightarrow{x : A_{ij} \left\{\overrightarrow{B/X}\right\}}^{j} \vdash^{\Theta}_{\mathcal{G}} v_i : A'_i \left\{\overrightarrow{B/X}\right\}}^{i}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{x}] \Rightarrow v_i}^{i}\right\} : \mathsf{G}(\overrightarrow{B})}\ \mathsf{G}I$$

$$\frac{\Theta \vdash_{\mathcal{G}} \mathsf{G}(\overrightarrow{B}) : \mathcal{S} \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v : \mathsf{G}(\overrightarrow{B}) \quad \overrightarrow{\Gamma \vdash^{\Theta}_{\mathcal{G}} v_j : A_{ij} \left\{\overrightarrow{B/X}\right\}}^{j}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v\ \ \mathsf{O}_i[\overrightarrow{v_j}^{j}] : A'_i}\ \mathsf{G}E_{\mathsf{O}_i}$$

FIGURE 9.3. Natural deduction typing rules for simple (co-)data.

Given **data** $\mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}$ **where** $\overrightarrow{\mathsf{K}_i : \overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y{:}l_{ij}}^{j}} \mathsf{F}(\overrightarrow{X})}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{\Theta \vdash_{\mathcal{G}} B' : l_{ij} \left\{\overrightarrow{B/X}\right\}}^{j} \quad \overrightarrow{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A_{ij} \left\{\overrightarrow{B'/Y}, \overrightarrow{B/X}\right\}}^{j}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{K}_i(\overrightarrow{B'}, \vec{v}) : \mathsf{F}(\vec{B})}\ \mathsf{F}I_{\mathsf{K}_i}$$

$$\frac{\Theta \vdash_{\mathcal{G}} \mathsf{F}(\vec{B}) : \mathcal{S} \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v : \mathsf{F}(\vec{B}) \quad \overrightarrow{\Gamma, \overrightarrow{x : A_{ij} \left\{\overrightarrow{B/X}\right\}}^{j} \vdash^{\Theta, \overrightarrow{Y{:}l_{ij}}^{j}}_{\mathcal{G}} v_i : C}^{i}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l_{ij}}^{j}, \vec{x}) \Rightarrow v_i}^{i} : C}\ \mathsf{F}E$$

Given **codata** $\mathsf{G}(\overrightarrow{X{:}k}) : \mathcal{S}$ **where** $\overrightarrow{\mathsf{O}_i : \overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\vec{X}) \vdash^{\overrightarrow{Y{:}l_{ij}}^{j}} A'_i : \mathcal{R}_i}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{\Gamma, \overrightarrow{x : A_{ij} \left\{\overrightarrow{B/X}\right\}}^{j} \vdash^{\Theta, \overrightarrow{Y{:}l_{ij}}^{j}}_{\mathcal{G}} v_i : A'_i \left\{\overrightarrow{B/X}\right\}}^{i}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \lambda \left\{ \overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l_{ij}}^{j}, \vec{x}] \Rightarrow v_i}^{i} \right\} : \mathsf{G}(\vec{B})}\ \mathsf{G}I$$

$$\frac{\Theta \vdash_{\mathcal{G}} \mathsf{G}(\vec{B}) : \mathcal{S} \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v : \mathsf{G}(\vec{B}) \quad \overrightarrow{\Theta \vdash_{\mathcal{G}} B' : l_{ij} \left\{\overrightarrow{B/X}\right\}}^{j} \quad \overrightarrow{\Gamma \vdash^{\Theta}_{\mathcal{G}} v_j : A_{ij} \left\{\overrightarrow{B'/Y}, \overrightarrow{B/X}\right\}}^{j}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v\ \ \mathsf{O}_i[\overrightarrow{B'}^{j}, \overrightarrow{v_j}^{j}] : A'_i}\ \mathsf{G}E_{\mathsf{O}_i}$$

FIGURE 9.4. Natural deduction typing rules for higher-order (co-)data.

$$\begin{array}{rll}
(let_{\mathcal{T}}) & \mathbf{let}\, x = V \,\mathbf{in}\, v \succ_{let_{\mathcal{T}}} v\,\{V/x\} & (V \in \mathit{Value}_{\mathcal{T}}) \\
(\eta_{let\mathcal{T}}) & \mathbf{let}\, x = v \,\mathbf{in}\, E[x] \succ_{\eta_{let\mathcal{T}}} E[v] & (E \in \mathit{CoValue}_{\mathcal{T}}, x \notin FV(E)) \\
(cc_{\mathcal{T}}) & E[\mathbf{let}\, x = v' \,\mathbf{in}\, v] \succ_{cc_{\mathcal{T}}} \mathbf{let}\, x = v' \,\mathbf{in}\, E[v] & (\Box \neq E \in \mathit{CoValue}_{\mathcal{T}}, x \notin FV(E)) \\
(cc_{\mathcal{T}}) & E\begin{bmatrix} \mathbf{case}\, v' \,\mathbf{of} \\ \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}) \Rightarrow v} \end{bmatrix} \succ_{cc_{\mathcal{T}}} \begin{matrix} \mathbf{case}\, v' \,\mathbf{of} \\ \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}) \Rightarrow E[v]} \end{matrix} &
\end{array}$$

FIGURE 9.5. A core parametric theory for the natural deduction calculus.

co-values are evaluation contexts with some additional properties: evaluation contexts in general are *stationary*, but co-values are *mobile*.

With the concept of λlet evaluation strategies in mind, lets look at the strategy-parametric rewriting rules for the λlet-calculus. The core rewriting rules, which correspond to the core theory of the $\mu\tilde{\mu}$-calculus, are given in Figure 9.5. These rules are responsible for interpreting **let** expressions: the $let_{\mathcal{T}}$ rule substitutes a **let**-bound $\mathcal{T}$-value for the bound variable, and the $\eta_{let\mathcal{T}}$ rule eliminates a trivial **let** expression of the form $\mathbf{let}\, x = v \,\mathbf{in}\, E[x]$ which introduces a name only to use it exactly once in the eye of a $\mathcal{T}$-co-value. The core theory also includes *commuting conversions* $cc_{\mathcal{T}}$ which push co-values inside of the block structures of **let** and **case** expressions. Intuitively, in the term $E[\mathbf{let}\, x = v' \,\mathbf{in}\, v]$, the result of v is passed to the co-value E, however the two are separated by an intermediate **let**. A $cc_{\mathcal{T}}$ reduction is thus needed to push the co-value inward and bring the question E in contact with the answer v as in $\mathbf{let}\, x = v' \,\mathbf{in}\, E[v]$. The same situation happens with a **case** in place of a **let**, so there is a commuting conversion for **case**, too. Unfortunately, this means that the core λlet theory must know about and manipulate language constructs revolving around data types, unlike the core theory of the $\mu\tilde{\mu}$ sequent calculus which made no assumptions about specific types.

Next, we have the rewriting rules for data and co-data in the natural deduction λlet-calculus. First are the untyped and strategy-parameterized β and ς laws in Figure 9.6, which mimic the similar β and ς in the $\mu\tilde{\mu}$-calculus. The β laws generalize the β laws for the λ-calculus from Chapter II to accomodate arbitrary data and co-data types and arbitrary substitution strategies. The ς lift laws are necessary to keep evaluation moving forward when non-values are found in unfortunate contexts.

$$(\beta_{\mathcal{T}}) \qquad \begin{array}{c}\textbf{case}\ \mathsf{K}_i(\vec{B}, \vec{V})\ \textbf{of} \\ \hline \overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v_i}^{\,i}\end{array} \succ_{\beta_{\mathcal{T}}} v_i \left\{\overrightarrow{V/x}, \overrightarrow{B/Y}\right\}$$

$$(\beta_{\mathcal{T}}) \qquad \lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v_i}^{\,i}\right\}\ \mathsf{O}_1[\vec{B}, \vec{V}] \succ_{\beta_{\mathcal{T}}} v_i \left\{\overrightarrow{V/x}, \overrightarrow{B/Y}\right\}$$

$$(\varsigma_{\mathcal{T}}) \qquad \mathsf{K}_i(\vec{B}, \vec{V}, v', \vec{v}) \succ_{\varsigma_{\mathcal{T}}} \begin{array}{l}\textbf{let}\, x = v' \\ \textbf{in}\ \mathsf{K}_i(\vec{B}, \vec{V}, x, \vec{v})\end{array} \qquad (v' \notin \mathit{Value}_{\mathcal{T}}, x \text{ fresh})$$

$$(\varsigma_{\mathcal{T}}) \qquad V'\ \ \mathsf{O}[\vec{B}, \vec{V}, v', \vec{v}] \succ_{\varsigma_{\mathcal{T}}} \begin{array}{l}\textbf{let}\, x = v' \\ \textbf{in}\, V'\ \ \mathsf{O}[\vec{B}, \vec{V}, x, \vec{v}]\end{array} \qquad (v' \notin \mathit{Value}_{\mathcal{T}}, x \text{ fresh})$$

FIGURE 9.6. The untyped parametric $\beta\varsigma$ laws for arbitrary data and co-data types.

For example, call-by-value λ-calculi often have issues with getting stuck prematurely on open terms, where evaluation should still continue even though the value of everything isn't known yet. For example, in the open λ-calculus term $(\lambda x.\lambda y.x)\ (f\ 1)\ 2$, plain call-by-value β-reduction is stuck because $f\ 1$ is not a value and cannot be substituted for x even though the result of the term must be $f\ 1$ for any value of f. However, the ς rules can lift the inconveniently-placed $f\ 1$ out of the way, letting reduction proceed as follows:

$$\begin{aligned}(\lambda x.\lambda y.x)\ (f\ 1)\ 2 &\to_{\varsigma_{\mathcal{V}}} \textbf{let}\, z = f\ 1\, \textbf{in}\, (\lambda x.\lambda y.x)\ z\ 2 \\ &\to_{\beta_{\mathcal{V}}} \textbf{let}\, z = f\ 1\, \textbf{in}\, (\lambda y.z)\ 2 \\ &\to_{\beta_{\mathcal{V}}} \textbf{let}\, z = f\ 1\, \textbf{in}\, z \\ &\to_{\eta_{let\mathcal{V}}} f\ 1\end{aligned}$$

We also have the typed and strategy-independent β and η laws in Figure 9.6. The β laws work for any evaluation strategy by binding unevaluated sub-terms in **let** expressions, and the η laws expand terms based on their type. Note that, as in the $\mu\tilde{\mu}$-calculus, the η law for co-data types acts on variables, but the more commonly seen generalization to values is derivable with the help of the core theory for **let** :

$$V : \mathsf{G}(\vec{C}) =_{let_{\mathcal{T}}} \textbf{let}\, y = V\, \textbf{in}\, y$$

$$
\begin{array}{ll}
(\beta^{\mathsf{F}}) & \textbf{case } \mathsf{K}_i(\vec{B}, \vec{v})\, \textbf{of}\, \overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v_i} \succ_{\beta^{\mathsf{F}}} \textbf{let}\, \overrightarrow{x = v_i}\, \textbf{in}\, v_i \left\{\overrightarrow{B/Y}\right\} \\
(\beta^{\mathsf{G}}) & \lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v_i}\right\}\ \mathsf{O}_i[\vec{B}, \vec{v}] \succ_{\beta^{\mathsf{G}}} \textbf{let}\, \overrightarrow{x = v}\, \textbf{in}\, v_i \left\{\overrightarrow{B/Y}\right\} \\
(\eta^{\mathsf{F}}) & v : \mathsf{F}(\vec{C}) \prec_{\eta^{\mathsf{F}}} \textbf{case}\, v\, \textbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow \mathsf{K}(\overrightarrow{Y{:}l}, \vec{x})} \\
(\eta^{\mathsf{G}}) & y : \mathsf{G}(\vec{C}) \prec_{\eta^{\mathsf{G}}} \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow y\ \ \mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}]}\right\}
\end{array}
$$

FIGURE 9.7. The typed $\beta\eta$ laws for declared data and co-data types.

$$
\begin{aligned}
V \in \mathit{Value}_{\mathcal{V}} &::= x \mid \mathsf{K}(\vec{A}, \vec{V}) \mid \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{X{:}l}, \vec{x}] \Rightarrow v}\right\} \\
E \in \mathit{CoValue}_{\mathcal{V}} &::= F \\
D \in \mathit{EvalCxt}_{\mathcal{V}} &::= E
\end{aligned}
$$

FIGURE 9.8. Call-by-value ($\mathcal{V}$) strategy in natural deduction.

$$
\begin{aligned}
&=_{\eta^{\mathsf{G}}} \textbf{let}\, y = V\, \textbf{in}\, \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow y\ \ \mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}]}\right\} \\
&=_{let_{\mathcal{T}}} \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow V\ \ \mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}]}\right\}
\end{aligned}
$$

where the definition of capture-avoiding substitution enforces the side condition that the variables $\vec{Y}$ and $\vec{x}$ are not free in V. In this sense, the strategy-independent η laws in Figure 9.7 also generalize the η laws for the λ-calculus from Chapter II.

Let's now consider some example evaluation strategies, corresponding to the ones we defined for the $\mu\tilde{\mu}$ sequent calculus. First, we have the call-by-value strategy $\mathcal{V}$ shown in Figure 9.8, which says that only variables, data structures made from values, and co-data objects are values. However, all frame contexts are co-values, which also exactly spell out the set of evaluation contexts. This definition essentially follows the normal notion of values and evaluation contexts in the call-by-value λ-calculus, except that evaluation does not descend into data structures (like pairs) or the arguments of co-data observations (like function calls). Instead, the ς rules lift unevaluated components out of these contexts and bind them to a variable with a **let** expression which *is* a co-value so evaluation can continue on them. Next, we have the call-by-name strategy $\mathcal{N}$ shown in Figure 9.8, which says that every term is a value, but

$$
\begin{aligned}
V \in \mathit{Value}_{\mathcal{N}} &::= v \\
E \in \mathit{CoValue}_{\mathcal{N}} &::= \square \mid \mathbf{case}\, E\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{X{:}l}, \vec{x}) \Rightarrow v} \mid E\ \ \mathsf{O}[\vec{A}, \vec{v}] \\
D \in \mathit{EvalCxt}_{\mathcal{N}} &::= E
\end{aligned}
$$

FIGURE 9.9. Call-by-name ($\mathcal{N}$) strategy in natural deduction.

only the empty context, a **case** on a co-value, and an observation on a co-value are co-values. This evaluation strategy more closely matches the call-by-name λ-calculus, where every term is substitutable and evaluation contexts, which are exactly co-values, are only those contexts which force an answer to be given for computation to continue.

Finally, we have the most subtle strategy of the three: the call-by-need strategy $\mathcal{LV}$ shown in Figure 9.10. Note how the values of call-by-need are exactly the values of call-by-value. However, the co-values of call-by-need lie in between call-by-value and call-by-name. In particular, every call-by-name co-value is a call-by-need co-value, but there are extra co-values of the form $\mathbf{let}\, x = E\, \mathbf{in}\, D[E'[x]]$, where the evaluation context D can include extra **let**-bindings around $E'[x]$. Note that this is the first evaluation strategy with an interesting difference between co-values and evaluation contexts: evaluation contexts can wrap a co-value with extra **let**-bindings as in $\mathbf{let}\, x_1 = v_1\, \mathbf{in} \ldots \mathbf{let}\, x_n = v_n\, \mathbf{in}\, E$. This is because those bindings are delayed until their value is needed, as in the call-by-need λ-calculus. For example, if we have the term $\mathbf{let}\, x = 1 + 1\, \mathbf{in}\, v$, the right-hand-side of $x = 1 + 1$ is delayed, and instead v is evaluated in the context $\mathbf{let}\, x = 1 + 1\, \mathbf{in}\, v$. If it happens that v reduces to a term that needs x, that is to say $v \mapsto\!\!\!\!\rightarrow D[E[x]]$, then $\mathbf{let}\, x = \square\, \mathbf{in}\, D[E[x]]$ is an evaluation context, so that $1 + 1$ is evaluated and substituted for x as in:

$$
\begin{aligned}
\mathbf{let}\, x = 1 + 1\, \mathbf{in}\, v &\mapsto\!\!\!\!\rightarrow \mathbf{let}\, x = 1 + 1\, \mathbf{in}\, D[E[x]] \\
&\mapsto \mathbf{let}\, x = 2\, \mathbf{in}\, D[E[x]] \\
&\mapsto D[E[2]]\, \{2/x\}
\end{aligned}
$$

However, the bindings are not mobile; they should not be pushed inward by commuting conversions like co-values are.

$$\begin{aligned}
V \in \mathit{Value}_{\mathcal{LV}} &::= x \mid \mathsf{K}(\vec{A}, \vec{V}) \mid \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{X{:}l}, \vec{x}] \Rightarrow v}\right\} \\
E \in \mathit{CoValue}_{\mathcal{LV}} &::= \square \mid \mathbf{case}\, E \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{X{:}l}, \vec{x}) \Rightarrow v} \mid E \;\; \mathsf{O}[\vec{A}, \vec{V}] \mid \mathbf{let}\, x = E \,\mathbf{in}\, D[E'[x]] \\
D \in \mathit{EvalCxt}_{\mathcal{LV}} &::= E \mid \mathbf{let}\, x = v \,\mathbf{in}\, D
\end{aligned}$$

FIGURE 9.10. Call-by-need ($\mathcal{LV}$) strategy in natural deduction.

Finally, we can combine multiple evaluation strategies within a single program using as similar technique as in the $\mu\tilde{\mu}$ sequent calculus. That is to say, evaluation strategies can be combined by taking the disjoint union of the respective substitution strategies and composing together each of their evaluation contexts to get a single set of operational contexts. The disjointness of the union can be regulated by kinds, as a looser form of typing discipline as shown in Figure 9.11. As before, the statement $v :: \mathcal{T}$ intuitively means that $v : A$ and $A : \mathcal{T}$ for some unknown type A. We can then disjointly union several substitution strategies $\vec{T}$ based on kinds by associating each strategy with a kind, and distinguishing values based on the kind of their output and co-values based on the kind of their input. That is to say, the combined set of values $\mathit{Value}_{\vec{\mathcal{T}}}$ contains any value $V \in \mathit{Value}_{\mathcal{T}_i}$ such that $v :: \mathcal{T}_i$. In contrast, the combined set of co-values $\mathit{CoValue}_{\vec{\mathcal{T}}}$ contains any co-value $E \in \mathit{CoValue}_{\mathcal{T}_i}$ such that $E[v] :: \mathcal{T}_j$ for all $v :: \mathcal{T}_i$. For example, to combine the three strategies above, we would get the following composite strategy:

$$\frac{V :: \mathcal{V} \quad V \in \mathit{Value}_{\mathcal{V}}}{V \in \mathit{Value}_{\mathcal{V},\mathcal{N},\mathcal{LV}}} \qquad \frac{V :: \mathcal{N} \quad V \in \mathit{Value}_{\mathcal{N}}}{V \in \mathit{Value}_{\mathcal{V},\mathcal{N},\mathcal{LV}}} \qquad \frac{V :: \mathcal{LV} \quad V \in \mathit{Value}_{\mathcal{LV}}}{V \in \mathit{Value}_{\mathcal{V},\mathcal{N},\mathcal{LV}}}$$

$$\frac{E \in \mathit{CoValue}_{\mathcal{V}} \quad \forall v :: \mathcal{V}.\exists \mathcal{R}.E[v] :: \mathcal{R}}{E \in \mathit{CoValue}_{\mathcal{V},\mathcal{N},\mathcal{LV}}}$$

$$\frac{E \in \mathit{CoValue}_{\mathcal{N}} \quad \forall v :: \mathcal{N}.\exists \mathcal{R}.E[v] :: \mathcal{R}}{E \in \mathit{CoValue}_{\mathcal{V},\mathcal{N},\mathcal{LV}}}$$

$$\frac{E \in \mathit{CoValue}_{\mathcal{LV}} \quad \forall v :: \mathcal{LV}.\exists \mathcal{R}.E[v] :: \mathcal{R}}{E \in \mathit{CoValue}_{\mathcal{V},\mathcal{N},\mathcal{LV}}}$$

$$\frac{E \in \mathit{CoValue}_{\mathcal{V},\mathcal{N},\mathcal{LV}}}{E \in \mathit{EvalCxt}_{\mathcal{V},\mathcal{N},\mathcal{LV}}} \qquad \frac{D \in \mathit{EvalCxt}_{\mathcal{V},\mathcal{N},\mathcal{LV}}}{\mathbf{let}\, x :: \mathcal{LV} = v \,\mathbf{in}\, D \in \mathit{EvalCxt}_{\mathcal{V},\mathcal{N},\mathcal{LV}}}$$

$$\mathit{Judgement} ::= \Gamma \vdash^{\Theta}_{\mathcal{G}} v :: A$$

Core kinding rules:

$$\frac{}{\Gamma, x :: \mathcal{T} \vdash^{\Theta}_{\mathcal{G}} x :: \mathcal{T}}\ \mathit{Var} \qquad \frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v :: \mathcal{T} \quad \Gamma, x :: \mathcal{T} \vdash^{\Theta}_{\mathcal{G}} v' :: \mathcal{R}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{let}\, x = v\, \mathbf{in}\, v' :: \mathcal{R}}\ \mathit{Let}$$

Given $\mathbf{data}\, \mathsf{F}(\overrightarrow{X : k}) : \mathcal{S}\, \mathbf{where}\, \overrightarrow{\mathsf{K}_i : \overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \vdash^{\overrightarrow{Y{:}l_{ij}}^{j}} \mathsf{F}(\overrightarrow{X})}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{\Gamma \vdash_{\mathcal{G}} v :: \mathcal{T}_{ij}}^{j}}{\Gamma \vdash_{\mathcal{G}} \mathsf{K}_i(\overrightarrow{B'}, \vec{v}) :: \mathcal{S}}\ \mathsf{F}I_{\mathsf{K}_i} \qquad \frac{\Gamma \vdash_{\mathcal{G}} v :: \mathcal{S} \quad \overrightarrow{\Gamma, \overrightarrow{x :: \mathcal{T}_{ij}}^{j} \vdash_{\mathcal{G}} v_i :: \mathcal{R}}^{i}}{\Gamma \vdash_{\mathcal{G}} \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l_{ij}}^{j}, \vec{x}) \Rightarrow v_i}^{i} :: \mathcal{R}}\ \mathsf{F}E$$

Given $\mathbf{codata}\, \mathsf{G}(\overrightarrow{X{:}k}) : \mathcal{S}\, \mathbf{where}\, \overrightarrow{\mathsf{O}_i : \overrightarrow{A_{ij} : \mathcal{T}_{ij}}^{j} \mid \mathsf{G}(\overrightarrow{X}) \vdash^{\overrightarrow{Y{:}l_{ij}}^{j}} A'_i : \mathcal{R}_i}^{i} \in \mathcal{G}$, we have the rules:

$$\frac{\overrightarrow{\Gamma, \overrightarrow{x : \mathcal{T}_{ij}}^{j} \vdash_{\mathcal{G}} v_i :: \mathcal{R}_i}^{i}}{\Gamma \vdash_{\mathcal{G}} \lambda \left\{ \overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l_{ij}}^{j}, \vec{x}] \Rightarrow v_i}^{i} \right\} :: \mathcal{S}}\ \mathsf{G}I \qquad \frac{\Gamma \vdash_{\mathcal{G}} v :: \mathcal{S} \quad \overrightarrow{\Gamma \vdash_{\mathcal{G}} v_j :: \mathcal{T}_{ij}}^{j}}{\Gamma \vdash_{\mathcal{G}} v\ \ \mathsf{O}_i[\overrightarrow{B'}^{j}, \vec{v_j}^{j}] :: \mathcal{R}_i}\ \mathsf{G}E_{\mathsf{O}_i}$$

FIGURE 9.11. Type-agnostic kind system for multi-kinded natural deduction terms.

Note that, the statement of a combination of evaluation strategies is not as clean in the λlet natural deduction calculus as it was in the $\mu\tilde{\mu}$ sequent calculus because the term-heavy syntax of λlet makes it more indirect to talk about concepts such as co-values.

Remark 9.1. It is worth considering if the η laws in Figure 9.7 really say all that needs to be said about extensionality for data types. For example, the instance of the η law for sum types $A \oplus B$ is

$$\begin{aligned} v : A \oplus B =_{\eta^\oplus} \ & \mathbf{case}\, v\, \mathbf{of} \\ & \quad \iota_1\,(x) \Rightarrow \iota_1\,(x) \\ & \quad \iota_2\,(y) \Rightarrow \iota_2\,(y) \end{aligned}$$

and after all, there are apparently much stronger extensionality laws for sums, like

$$\begin{aligned} C[v : A \oplus B] = \ & \mathbf{case}\, v\, \mathbf{of} \\ & \quad \iota_1\,(x) \Rightarrow C[\iota_1\,(x)] \\ & \quad \iota_2\,(y) \Rightarrow C[\iota_2\,(y)] \end{aligned}$$

which generalizes $\eta^\oplus$ so that the term $v : A \oplus B$ may appear in any context C.[1] Unfortunately, this strong sum η law is deeply troublesome when faced with computational effects like nontermination. For example, if we apply the strong η law for sums with $C = \lambda x.x\ \Box$ and $v = \Omega : A \oplus B$, where Ω is a term which loops forever without returning a result, then the stronger $\eta^\oplus$ law is completely unsound with respect to contextual equivalence, since

$$\lambda z.z\ \Omega \ncong \mathbf{case}\, \Omega\, \mathbf{of}\, \iota_1\,(x) \Rightarrow \lambda z.z\ \iota_1\,(x) \mid \iota_2\,(y) \Rightarrow \lambda z.z\ \iota_2\,(y) \cong \Omega$$

Thus, the exceptionally strong version of $\eta^\oplus$ only makes sense in a pure and normalizing language, where everything terminates and all terms evaluate to some result.

Dealing with a strong extensionality law like this, which places very strict requirements on the language like strong normalization that can only be deep properties of the language as a whole, is difficult to handle directly. As an alternative

[1] This strong sum law is sometimes also written in terms of a substitution $v' \{v/z\}$ where v can occur in many places instead of the context $C[v]$ where v occurs in exactly one place, but these amount to the same thing since C might be $\mathbf{let}\, z = \Box\, \mathbf{in}\, v'$ so that $C[v] = \mathbf{let}\, z = v\, \mathbf{in}\, v'$.

approach, Munch-Maccagnoni & Scherer (2015) propose the use of polarization to tame the strong sum extensionality law to make it more manageable without loosing anything important. In particular, the polarization hypothesis suggests that call-by-value is the most fitting evaluation strategy for a data type like $A \oplus B$, and so in the strong $\eta^{\oplus}$ would be more appropriate to restrict the term in question to be a (call-by-value) value as in:

$$C[V : A \oplus B] = \mathbf{case}\, V\, \mathbf{of}\, \iota_1\,(x) \Rightarrow C[\iota_1\,(x)] \mid \iota_2\,(y) \Rightarrow C[\iota_2\,(y)]$$

This equation does not suffer from the same troubles when effects like nontermination are introduced: an infinitely looping term is never a value, so the above extensionality law does not cause the same sort of counter-example. In other words, this restricted version of the strong extensionality law for sums is sound even in the presence of effects. And as Munch-Maccagnoni & Scherer note, if the language happens to be strongly normalizing, then every (closed) term reduces to a value, anyway, and so the unrestricted sum extensionality law can be derived after the fact as a deeper property of the language.

So where does this leave our simplistic treatment of extensionality for data types taken here? As it turns out, the strong sum law is derivable from the simplistic $\eta^{\oplus}$ law in the equational theory with the help of commuting conversions. First, note that Munch-Maccagnoni & Scherer's extensionality law for call-by-value sums is derivable as follows:

$$\begin{aligned}
&\begin{array}{l}\mathbf{case}\, V\, \mathbf{of} \\ \quad \iota_1\,(x) \Rightarrow C[\iota_1\,(x)] \\ \quad \iota_2\,(y) \Rightarrow C[\iota_2\,(y)]\end{array}
&&=_{let_{\mathcal{V}}} \begin{array}{l}\mathbf{case}\, V\, \mathbf{of} \\ \quad \iota_1\,(x) \Rightarrow \mathbf{let}\, z = \iota_1\,(x)\, \mathbf{in}\, C[z] \\ \quad \iota_2\,(y) \Rightarrow \mathbf{let}\, z = \iota_2\,(y)\, \mathbf{in}\, C[z]\end{array} \\
& &&=_{cc_{\mathcal{V}}} \begin{array}{l}\mathbf{let}\, z = \begin{array}[t]{l}\mathbf{case}\, V\, \mathbf{of} \\ \quad \iota_1\,(x) \Rightarrow \iota_1\,(x) \\ \quad \iota_2\,(y) \Rightarrow \iota_2\,(y)\end{array} \\ \mathbf{in}\, C[z]\end{array} \\
& &&=_{\eta^{\oplus}} \mathbf{let}\, z = V\, \mathbf{in}\, C[z] \\
& &&=_{let_{\mathcal{V}}} C[V]
\end{aligned}$$

$$\dfrac{\Gamma \vdash_{\mathcal{G}}^{\Theta} v_0 : A\ 0 \quad \Gamma, x : A\ j \vdash_{\mathcal{G}}^{\Theta, j:\mathsf{Ix}} v_1 : A\ (j+1)}{\Gamma \vdash_{\mathcal{G}}^{\Theta} \lambda\{\ @\ 0{:}\mathsf{Ix} \Rightarrow v_0 \mid\ @_x\ (j{+}1){:}\mathsf{Ix} \Rightarrow v_1\} : \forall_{\mathsf{Ix}}(A)}\ \forall_{\mathsf{Ix}} R_{rec}$$

$$\dfrac{\Theta \vdash_{\mathcal{G}} \exists_{\mathsf{Ix}}(A) : \mathcal{S} \quad \Gamma \vdash_{\mathcal{G}}^{\Theta} v : \exists_{\mathsf{Ix}}(A) \quad \Gamma, x : A\ 0 \vdash_{\mathcal{G}}^{\Theta} v_0 : C \quad \Gamma, x : A\ (j+1) \vdash_{\mathcal{G}}^{\Theta, j:\mathsf{Ix}} v_1 : A\ j}{\Gamma \vdash_{\mathcal{G}}^{\Theta} \mathbf{loop}\, v\, \mathbf{of}\ 0{:}\mathsf{Ix}\ @\ x \Rightarrow v_0 \mid (j{+}1) : \mathsf{Ix}\ @\ x \Rightarrow v_1 : C}\ \exists_{\mathsf{Ix}} L_{rec}$$

$$\dfrac{\Gamma, x : \forall_{<\mathsf{Ord}}(j, A) \vdash_{\mathcal{G}}^{\Theta, j<N} v : A\ j}{\Gamma \vdash_{\mathcal{G}}^{\Theta} \lambda\{\ @_x\ j{<}N \Rightarrow v\} : \forall_{<\mathsf{Ord}}(N, A)}\ \forall_{<\mathsf{Ord}} R_{rec}$$

FIGURE 9.12. The pure, recursive size abstractions in natural deduction.

This derivation makes two key observations about the call-by-value equational laws in the core theory: (1) every context C is equivalent to a $\mathcal{V}$ co-value evaluation context $\mathbf{let}\, z = \square\, \mathbf{in}\, C[z]$, and (2) this co-value commutes with a **case** expression. In total, we can effectively transport any such context out of the branches of a **case**, leaving just a trivial **case** expression that is handled by the simple $\eta^{\oplus}$ law. We can then deduce that the simple $\beta\eta$ equational theory for (co-)data types includes the strong sum law from Munch-Maccagnoni & Scherer's observation: so long as the language is strongly normalizing, then the strong sum extensionality law is sound with respect to contextual equivalence by the above derivation since every closed term is equivalent to a value. *End remark* 9.1.

Well-founded recursion

To incorporate well-founded recursion into the natural deduction λlet-calculus, we only need to allow for the same forms of recursively-defined types as in the $\mu\tilde{\mu}$-calculus in Chapter VI as well as the recursive abstraction over size indexes. Specifically, the typing and rewriting rules for the extra recursive terms for size abstractions are shown in Figures 9.12 and 9.13. Intuitively, the objects of $\forall_{\mathsf{Ix}}(A)$ are stepwise loops that can return any $A\ N$ by counting up from 0 and using the previous instances of itself, whereas we can write looping case expressions over values of $\exists_{\mathsf{Ix}}(A)$ to count down from any $A\ N$ to 0. Similarly, values of $\forall_{<\mathsf{Ord}}(N, A)$ are self-referential objects that always behave the same no matter the number of recursive invocations. Curiously though, the recursive forms for $\exists_{<\mathsf{Ord}}(N, A)$ are conspicuously missing from the functional calculus. In essence, the recursive form for $\exists_{<\mathsf{Ord}}(N, A)$ is a case expression that introduces a continuation variable for the recursive path out of the expression in addition to the

$$(\beta^{\forall_{\mathsf{Ix}}}_{\mathcal{T}}) \qquad \lambda\left\{\begin{array}{r} @\,[0{:}\mathsf{Ix}] \Rightarrow v_0 \\ @_x\,[j{+}1{:}\mathsf{Ix}] \Rightarrow v_1 \end{array}\right\} @\, 0 \succ_{\beta^{\forall_{\mathsf{Ix}}}_{\mathcal{T}}} v_0$$

$$(\beta^{\forall_{\mathsf{Ix}}}_{\mathcal{T}}) \quad \lambda\left\{\begin{array}{r} @\,[0{:}\mathsf{Ix}] \Rightarrow v_0 \\ @_x\,[j{+}1{:}\mathsf{Ix}] \Rightarrow v_1 \end{array}\right\} @\, [M+1] \succ_{\beta^{\forall_{\mathsf{Ix}}}_{\mathcal{T}}} \begin{array}{l} \mathbf{let}\, x = \lambda\left\{\begin{array}{r} @\,[0{:}\mathsf{Ix}] \Rightarrow v_0 \\ @_x\,[j{+}1{:}\mathsf{Ix}] \Rightarrow v_1 \end{array}\right\} @\, M \\ \mathbf{in}\, v_1\,\{M/j\} \end{array}$$

$$(\beta^{\exists_{\mathsf{Ix}}}_{\mathcal{T}}) \qquad \begin{array}{l} \mathbf{loop}\, V\ @\ 0\, \mathbf{of} \\ \quad x\ @\ 0{:}\mathsf{Ix} \Rightarrow v_0 \\ \quad x\ @\ (j{+}1{:}\mathsf{Ix}) \Rightarrow v_1 \end{array} \succ_{\beta^{\exists_{\mathsf{Ix}}}} v_0\,\{V/x\}$$

$$(\beta^{\exists_{\mathsf{Ix}}}_{\mathcal{T}}) \qquad \begin{array}{l} \mathbf{loop}\, V\ @\ (M+1)\, \mathbf{of} \\ \quad x\ @\ (0{:}\mathsf{Ix}) \Rightarrow v_0 \\ \quad x\ @\ (j{+}1{:}\mathsf{Ix}) \Rightarrow v_1 \end{array} \succ_{\beta^{\exists_{\mathsf{Ix}}}} \begin{array}{l} \mathbf{loop}\, v_1\,\{V/x, M/j\}\ @\ M\, \mathbf{of} \\ \quad x\ @\ (0{:}\mathsf{Ix}) \Rightarrow v_0 \\ \quad x\ @\ (j{+}1{:}\mathsf{Ix}) \Rightarrow v_1 \end{array}$$

$$(\nu^{\forall_{<\mathsf{Ord}}}) \qquad \lambda\{\ @_x\,[j{<}M] \Rightarrow v\} \succ_{\nu^{\forall_{<\mathsf{Ord}}}} \lambda\{@\,[i{<}M] \Rightarrow v\,\{\lambda\{x\ @\ [j{<}i] \Rightarrow v\}/x\}\}$$

FIGURE 9.13. The β and ν laws for recursion.

normal return path, effectively requiring a form of subtraction type $C - \exists_{<\mathsf{Ord}}(M, A)$ for smaller indices M to be useful. If we were to try to come up with a pure looping term for $\exists_{<\mathsf{Ord}}(N, A)$ in the λlet, we would end up with something like

$$\frac{\Theta \vdash_{\mathcal{G}} \exists_{<\mathsf{Ord}} N, A : \mathcal{S} \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} v : \exists_{<\mathsf{Ord}}(N, A) \quad \Gamma, x : A\ j \vdash^{\Theta, j<N}_{\mathcal{G}} v' : \exists_{<\mathsf{Ord}}(j, A)}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{loop}\, v\, \mathbf{of}\, x\ @\ j{<}N \Rightarrow v' : C}\ \exists_{<\mathsf{Ord}} L_{rec}$$

which is not a useful construct since there is no way to return a result of type C from this term, making it unobservable. So while $\exists_{<\mathsf{Ord}}$ can still be used to hide Ord indices, its recursive nature lies outside the pure functional paradigm. This follows the frequent situation where one of four classical principles gets lost in translation to intuitionistic settings. It occurs with De Morgan laws ($\neg(A \wedge B) \rightarrow (\neg A) \vee (\neg B)$ is not intuitionistically valid), the conjunctive and disjunctive connectives (⅋ requires multiple conclusions so it does not fit the pure mold), and here as well, and also in the orthogonality semantics of programs from Chapter VII where the space orthogonal to an intersection contains more than the union of the orthogonal parts.

Example 9.1. Intuitively, we can think of the values of $\forall_{\mathsf{Ix}}(A)$ as a dependently typed version of the recursion operator for natural numbers in Gödel's (1980) System T.

Indeed, we can encode such an operator using deep co-pattern matching as:

$$
\begin{aligned}
&rec : \forall X : \mathsf{Ix} \to \mathcal{R}.X\ 0 \to (\forall i : \mathsf{Ix}.X\ i \to X\ (i+1)) \to \forall i : \mathsf{Ix}.X\ i \\
&rec = \lambda\left\{\begin{array}{ll} @\, X \cdot x \cdot f\ @\ (0{:}\mathsf{Ix}) & \Rightarrow x \\ @\, X \cdot x \cdot f\ @_r\ (j{+}1{:}\mathsf{Ix}) & \Rightarrow f\ @\ j \cdot r \end{array}\right\}
\end{aligned}
$$

which desugars into the shallow co-patterns:

$$
\begin{aligned}
&rec : \forall X : \mathsf{Ix} \to \mathcal{R}.X\ 0 \to (\forall i : \mathsf{Ix}.X\ i \to X\ (i+1)) \to \forall i : \mathsf{Ix}.X\ i \\
&rec = \Lambda X.\lambda x.\lambda f.\lambda\{\ @\ (0{:}\mathsf{Ix}) \Rightarrow x \mid\ @_r\ (j{+}1{:}\mathsf{Ix}) \Rightarrow f\ @\ j \cdot r\}
\end{aligned}
$$

So essentially, we are using the natural number index to drive the recursion upward to compute some value, where the type of that returned value can depend on the number of steps in the chosen index. In a call-by-name setting, where we choose a maximal set of values so that V can be any term, then the behavior of rec implements the recursor: given that $rec\ X\ x\ f \twoheadrightarrow_{\beta_{\mathcal{N}}} rec_{X,x,f}$ we have

$$
rec_{X,x,f}\ @\ 0 \to_{\beta^{\forall_{\mathsf{Ix}}}_{\mathcal{N}}} x \qquad rec_{X,x,f}\ @\ (M+1) \to_{\beta^{\forall_{\mathsf{Ix}}}_{\mathcal{N}}} f\ @\ M \cdot (rec_{X,x,f}\ @\ M)
$$

Contra-posed, $\exists_{\mathsf{Ix}}(A)$ implements a dependently-typed, stepwise recursion in the other direction. The looping form breaks down a value depending on an arbitrary index N by counting down until that index reaches 0, finally returning some value which does *not* depend on the index. For instance, we can sum the values in any vector of numbers, $v : \mathsf{Vec}(N, \mathsf{ANat})$, where Vec and Nat are declared as in Chapter VI, in accumulator style by looping over the recursive structure $\exists i : \mathsf{Ix}\,.\, \mathsf{ANat} \otimes \mathsf{Vec}(i, \mathsf{ANat})$:[2]

$$
\begin{array}{l}
\mathbf{loop}\, N\ @\ ((0\ @\ \mathsf{Z}), v)\, \mathbf{of} \\
\quad \begin{array}{ll} (0{:}\mathsf{Ix})\ @\ (acc, \mathsf{Nil}) & \Rightarrow acc \\ (i{+}1{:}\mathsf{Ix})\ @\ (acc, \mathsf{Cons}(x, xs)) & \Rightarrow (x + acc, xs) \end{array}
\end{array}
$$

End example 9.1.

Example 9.2. Instead, values of $\forall_{<\mathsf{Ord}}$ are useful for representing stronger induction that recurses on deeply nested sub-structures. For example, we can convert a list

[2] Note, we assume an addition operator $+ : \mathsf{ANat} \to \mathsf{ANat} \to \mathsf{ANat}$.

$x_1, x_2, \ldots, x_n$, where List is declared the same as in Chapter VI, into a list of its adjacency pairs $(x_1, x_2), (x_3, x_4), \ldots, (x_{n-1}, x_n)$ by

$$\begin{aligned}
&\mathit{pairs}\ \ \mathsf{Nil} &&= \mathsf{Nil} \\
&\mathit{pairs}\ \ \mathsf{Cons}(x, ys) &&= \mathsf{Nil} \\
&\mathit{pairs}\ \ \mathsf{Cons}(x, \mathsf{Cons}(y, zs)) &&= \mathsf{Cons}((x, y), \mathit{pairs}\ zs)
\end{aligned}$$

where we silently drop the final element if the list is odd. The *pairs* function can be straightforwardly encoded using $\forall_{<\mathsf{Ord}}$ as:

$$\mathit{pairs} : \forall X : \star. \forall i < \infty.\, \mathsf{List}(i, X) \to \mathsf{List}(i, X \otimes X)$$

$$\mathit{pairs} = \Lambda X. \lambda \left\{ \begin{array}{l}
@_r\ i{<}\infty \Rightarrow \lambda x{:}\mathsf{List}(i, X).\, \mathbf{case}\, xs\, \mathbf{of} \\
\qquad \mathsf{Nil} \Rightarrow \mathsf{Nil} \\
\qquad \mathsf{Cons}^{j<i}(x{:}X, ys{:}\mathsf{List}(j, X)) \Rightarrow \mathbf{case}\, ys\, \mathbf{of} \\
\qquad\quad \mathsf{Nil} \Rightarrow \mathsf{Nil} \\
\qquad\quad \mathsf{Cons}^{k<j}(y{:}X, zs{:}\mathsf{List}(k, X)) \Rightarrow \mathsf{Cons}^{k}((x, y), (r\ @\ k) \cdot zs)
\end{array} \right\}$$

Note that the type of the recursive argument r is $\forall i' < i.\, \mathsf{List}(i', X) \to \mathsf{List}(i', X \otimes X)$. Thus, the recursive self-invocation $r\ @\ k : \mathsf{List}(k, X) \to \mathsf{List}(k, X \otimes X)$ is well-typed, since we learn that $j < i$ and $k < j$ by analyzing the Cons structure of the list, and can infer that $k < i$ by transitivity. *End example* 9.2.

Example 9.3. As a comparison between recursive data and co-data, consider the following two alternative methods of encoding branching structure.

$$\begin{array}{lll}
\mathbf{data}\, \mathsf{Tree}(i : \mathsf{Ix}, X : \mathcal{S}) : \mathcal{S}\, \mathbf{by} \text{ primitive recursion on } i & & \\
\mathbf{where}\, i = 0 & \mathsf{Leaf} : & \vdash \mathsf{Tree}(0, X) \mid \\
\mathbf{where}\, i = j + 1 & \mathsf{Branch} : & \mathsf{Tree}(j, X) : \mathcal{S}, X : \mathcal{S}, \mathsf{Tree}(j, X) : \mathcal{S} \vdash \mathsf{Tree}(j + 1, X) \mid
\end{array}$$

$$\begin{array}{lrl}
\mathbf{codata}\, \mathsf{River}(i : \mathsf{Ix}, X : \mathcal{S}) : \mathcal{S}\, \mathbf{by} \text{ primitive recursion on } i & & \\
\mathbf{where}\, i = j + 1 & \mathsf{ForkL} : & \mid \mathsf{River}(j + 1, X) \vdash \mathsf{River}(j, X) : \mathcal{S} \\
& \mathsf{Curr} : & \mid \mathsf{River}(j + 1, X) \vdash X : \mathcal{S} \\
& \mathsf{ForkR} : & \mid \mathsf{River}(j + 1, X) \vdash \mathsf{River}(j, X) : \mathcal{S}
\end{array}$$

The first is the data type $\mathsf{Tree}(i, X)$ defined by primitive recursion on i that represents balanced binary trees of exactly height i containing an X element in each internal Branch node. The second is the co-data type $\mathsf{River}(i, X)$ defined by primitive recursion on i that represents a branching stream which has a current element (accessed by Curr), a left fork (accessed by ForkL) and a right fork (accessed by ForkR). The index i in $\mathsf{River}(i, X)$ dictates how far we can flow down the forking river before it runs dry.

The two types are effectively different representations of the same information. From a $\mathsf{River}(i, X)$ of a given depth i, we can grow a $\mathsf{Tree}(i, X)$ of the same height i by flowing down each forking path in each branch as follows:

$$\mathit{grow} : \forall X : \mathcal{S}.\forall i : \mathsf{Ix}.\, \mathsf{River}(i, X) \to \mathsf{Tree}(i, X)$$

$$\mathit{grow} = \lambda\left\{\begin{array}{ll} @\,X\,@\,(0{:}\mathsf{Ix}) \cdot s & \Rightarrow \mathsf{Leaf} \\ @\,X\,@_f\,(j{+}1{:}\mathsf{Ix}) \cdot s & \Rightarrow \mathsf{Branch}\,(f \cdot (s\ \ \mathsf{ForkL}), s\ \ \mathsf{Curr}, f \cdot (s\ \ \mathsf{ForkR})) \end{array}\right\}$$

Conversely, we can chop down a $\mathsf{Tree}(i, X)$ of height i and have its elements flow down a $\mathsf{River}(i, X)$ of the same depth i by traversing each branch, keeping the branch's X element as the current element of the river, and having the left and right sides of the branch flow down the left and right forks of the river as follows:

$$\mathit{chop} : \forall X : \mathcal{S}.\forall i : \mathsf{Ix}.\, \mathsf{Tree}(i, X) \to \mathsf{River}(i, X)$$

$$\mathit{chop} = \lambda\left\{\begin{array}{lll} @\,X\,@_f\,(j{+}1{:}\mathsf{Ix}) \cdot \mathsf{Branch}(l, x, r) & \mathsf{ForkL} & \Rightarrow f \cdot l \\ @\,X\,@_f\,(j{+}1{:}\mathsf{Ix}) \cdot \mathsf{Branch}(l, x, r) & \mathsf{Curr} & \Rightarrow x \\ @\,X\,@_f\,(j{+}1{:}\mathsf{Ix}) \cdot \mathsf{Branch}(l, x, r) & \mathsf{ForkL} & \Rightarrow f \cdot r \end{array}\right\}$$

One way to interpret *chop* is that the ForkL, Curr, and ForkR observers from River form the constructors of a zipper (Huet, 1997) into Trees. Indeed, in the $\mu\tilde{\mu}$ sequent calculus, *chop* would be written as

$$\mathit{chop} = \mu\left(\begin{array}{ll} [X\ @\ (j{+}1{:}\mathsf{Ix})\ @_f\ \mathsf{Branch}(l, x, r) \cdot \mathsf{ForkL}[\beta_1]]\,. & \langle f \| l \cdot \beta_1 \rangle \\ [X\ @\ (j{+}1{:}\mathsf{Ix})\ @_f\ \mathsf{Branch}(l, x, r) \cdot \mathsf{Curr}[\alpha]]\,. & \langle x \| \alpha \rangle \\ [X\ @\ (j{+}1{:}\mathsf{Ix})\ @_f\ \mathsf{Branch}(l, x, r) \cdot \mathsf{ForkR}[\beta_2]]\,. & \langle f \| r \cdot \beta_2 \rangle \end{array}\right)$$

where a River observation like $\mathsf{ForkL}[\mathsf{ForkR}[\mathsf{ForkR}[\mathsf{ForkL}[\mathsf{Curr}[\alpha]]]]]$ is a first-class co-data structure detailing a path for *chop* to descend into a tree.

Both of these objects above were written using deep pattern matching, which we can desugar into shallow patterns as usual. The desugared version of *grow* is not terribly surprising, but *chop* is more interesting. In particular, by spelling out each case separately, it is clear that a 0 height tree is converted into a nullary river $\lambda\{\} : \mathsf{River}(0, X)$ that does not have to respond to any observation, because there are none in the 0 case.

$$grow = \Lambda X.\lambda\left\{\begin{array}{ll} @\ (0{:}\mathsf{Ix}) & \Rightarrow \lambda s.\,\mathsf{Leaf} \\ @_f\ (j{+}1{:}\mathsf{Ix}) & \Rightarrow \lambda s.\,\mathsf{Branch}\,(f \cdot (s\ \ \mathsf{ForkL}), s\ \ \mathsf{Curr}, f \cdot (s\ \ \mathsf{ForkR})) \end{array}\right\}$$

$$chop = \Lambda X.\lambda\left\{\begin{array}{ll} @\ (0{:}\mathsf{Ix}) & \Rightarrow \lambda t.\lambda\{\} \\ @_f\ (j{+}1{:}\mathsf{Ix}) & \Rightarrow \lambda t.\,\mathbf{case}\,t\,\mathbf{of}\ \mathsf{Branch}(l, x, r) \Rightarrow \lambda\left\{\begin{array}{ll} \mathsf{ForkL} & \Rightarrow f \cdot l \\ \mathsf{Curr} & \Rightarrow x \\ \mathsf{ForkR} & \Rightarrow f \cdot r \end{array}\right\} \end{array}\right\}$$

End example 9.3.

Natural Deduction versus Sequent Calculus

The natural deduction based λlet is heavily influenced by language features that have been developed in the $\mu\tilde{\mu}$ sequent calculus, but how do the two really compre? For example, $\mu\tilde{\mu}$ allows for the first-class treatment of control flow with its μ-abstractions, which are entirely missing from λlet, so clearly the intuitionistic λlet cannot directly correspond to *all* of the classical $\mu\tilde{\mu}$. Instead, the λlet-calculus corresponds to a well-established subset of $\mu\tilde{\mu}$ based off Gentzen's (1935a) intuitionistic LJ sequent calculus. LJ is exactly the same as LK that we saw in Chapter III, except that every sequent is limited to exactly *one* consequence. In terms of the $\mu\tilde{\mu}$-calculus, this corresponds a restriction to forcing exactly *one* free co-variable in every command and co-term (which have no active consequence), and *zero* free co-variables in every term (which already have an active consequence). It turns out this restriction is enough to establish a static and dynamic correspondence between the λlet-calculus based on natural deduction and the single-consequence $\mu\tilde{\mu}$-calculus based on the sequent calculus.

First, we consider how to translate between expressions of λlet and $\mu\tilde{\mu}$. The canonical, compositional translations are given in Figure 9.14. The translation $LJ[\![v]\!]$

converts λlet terms into $\mu\tilde{\mu}$ terms and $LJ[\![F]\!]$ converts λlet frames into $\mu\tilde{\mu}$ co-terms. Variables and introduction forms (i.e. data structures and co-data objects) translate directly to similar forms in the sequent calculus, whereas **let** and elimination forms (i.e. data **case** expressions and co-data observations) require the help of a μ and cut in the sequent calculus to represent the elimination. Going the other way, the $NJ[\![\_]\!]$ family of translations convert each $\mu\tilde{\mu}$ command and (co-)term into a λlet term. Also note that for commands and co-terms, $NJ[\![c]\!]_\alpha$ and $NJ[\![e]\!]_\alpha$ denotes the single free co-variable found in c and e. Terms translate directly to similar forms in natural deduction, whereas co-terms translate to contexts, with the singled-out co-variable α being the empty context, $\tilde{\mu}$-abstractions being a **let** context, and the left co-terms for (co-)data being elimination forms of (co-)data types. Finally, a cut $\langle v \| e \rangle$ is translated as plugging the translation of v, which is a term, into the translation of e, which is a context. Finally, we also extend the translation to substitution strategies pointwise, by saying that the values and co-values of a strategy in one language translate to the values and co-values of the other, which we close under reduction. In other words, given a λlet substitution strategy $\mathcal{T} = (\mathit{Value}_\mathcal{T}, \mathit{CoValue}_\mathcal{T})$, then the translation of $\mathcal{T}$ to $\mu\tilde{\mu}$ is

$$\begin{aligned} LJ[\![\mathcal{T}]\!] = (&\{v \in \mathit{Term} \mid \exists V \in \mathit{Value}_\mathcal{T}, LJ[\![V]\!] \twoheadrightarrow v\}, \\ &\{e \in \mathit{CoTerm} \mid \exists E \in \mathit{CoValue}_\mathcal{T}, LJ[\![E]\!] \twoheadrightarrow e\}) \end{aligned}$$

and similarly in the other direction, given a $\mu\tilde{\mu}$ substitution strategy $\mathcal{S} = (\mathit{Value}_\mathcal{S}, \mathit{CoValue}_\mathcal{S})$, the translation of $\mathcal{S}$ is

$$\begin{aligned} NJ[\![\mathcal{S}]\!] = (&\{v \in \mathit{Term} \mid \exists V \in \mathit{Value}_\mathcal{S}, LJ[\![V]\!] \twoheadrightarrow v\}, \\ &\{F \in \mathit{FrameCxt} \mid \exists E, \alpha \in \mathit{CoValue}_\mathcal{S}, \forall v \in Term, LJ[\![E]\!]_\alpha[v] \twoheadrightarrow F[v]\}) \end{aligned}$$

We can check that these translations preserve the semantics of the two languages. In terms of the static semantics, the two translations preserve types on the nose. That is to say, if we have a term $v : A$ in λlet then $LJ[\![v]\!] : A$ with exactly the same type A in $\mu\tilde{\mu}$, and if $v : A$ in $\mu\tilde{\mu}$ then $NJ[\![v]\!] : A$ in λlet. Note that this shows that the translations are direct, instead of indirect. For example, *continuation-passing style* (CPS) translations, which are an alternative to the kinds of translations in Figure 9.14, alter the interface of terms in a way seen in their types. That is to say, a CPS translation

Natural deduction to sequent calculus:

$$\begin{aligned}
LJ\llbracket x \rrbracket &\triangleq x \\
LJ\llbracket \mathbf{let}\, x = v \,\mathbf{in}\, v' \rrbracket &\triangleq \mu\alpha.\, \langle LJ\llbracket v \rrbracket \| \tilde{\mu}x.\, \langle LJ\llbracket v' \rrbracket \| \alpha \rangle\rangle \\
LJ\left\llbracket \mathsf{K}(\vec{B}, \vec{v}) \right\rrbracket &\triangleq \mathsf{K}^{\vec{B}}(\overrightarrow{LJ\llbracket v \rrbracket}) \\
LJ\left\llbracket \mathbf{case}\, v' \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} \right\rrbracket &\triangleq \mu\alpha.\, \left\langle LJ\llbracket v' \rrbracket \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y:l}}(\vec{x}).\langle LJ\llbracket v \rrbracket \| \alpha \rangle}\right] \right\rangle \\
LJ\left\llbracket \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\right\} \right\rrbracket &\triangleq \mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y:l}}[\vec{x}, \alpha].\langle LJ\llbracket v \rrbracket \| \alpha \rangle}^{i}\right) \\
LJ\left\llbracket v' \;\; \mathsf{O}[\vec{B}, \vec{v}] \right\rrbracket &\triangleq \mu\alpha.\, \left\langle LJ\llbracket v' \rrbracket \middle\| \mathsf{O}^{\vec{B}}[\overrightarrow{LJ\llbracket v \rrbracket}, \alpha] \right\rangle
\end{aligned}$$

$$\begin{aligned}
LJ\llbracket \square \rrbracket_{\alpha} &\triangleq \alpha \\
LJ\llbracket \mathbf{let}\, x = F \,\mathbf{in}\, v \rrbracket_{\alpha} &\triangleq LJ\llbracket F \rrbracket_{\alpha} \{\tilde{\mu}x.\, \langle LJ\llbracket v \rrbracket \| \alpha \rangle / \alpha\} \\
LJ\left\llbracket \mathbf{case}\, F \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} \right\rrbracket_{\alpha} &\triangleq LJ\llbracket F \rrbracket_{\alpha} \left\{\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y:l}}(\vec{x}).\langle LJ\llbracket v \rrbracket \| \alpha \rangle}\right] / \alpha\right\} \\
LJ\left\llbracket F \;\; \mathsf{O}[\vec{B}, \vec{v}] \right\rrbracket_{\alpha} &\triangleq LJ\llbracket F \rrbracket_{\alpha} \left\{\mathsf{O}^{\vec{B}}[LJ\llbracket \vec{v} \rrbracket, \alpha] / \alpha\right\}
\end{aligned}$$

Sequent calculus to natural deduction:

$$NJ\llbracket \langle v \| e \rangle \rrbracket_{\alpha} \triangleq NJ\llbracket e \rrbracket_{\alpha}[NJ\llbracket v \rrbracket]$$

$$\begin{aligned}
NJ\llbracket x \rrbracket &\triangleq x \\
NJ\llbracket \mu\alpha.c \rrbracket &\triangleq NJ\llbracket c \rrbracket_{\alpha} \\
NJ\left\llbracket \mathsf{K}^{\vec{B}}(\vec{v}) \right\rrbracket &\triangleq \mathsf{K}(\vec{B}, \overrightarrow{NJ\llbracket v \rrbracket}) \\
NJ\left\llbracket \mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y:l}}[\vec{x}, \alpha].c}\right) \right\rrbracket &\triangleq \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow NJ\llbracket c \rrbracket_{\alpha}}\right\}
\end{aligned}$$

$$\begin{aligned}
NJ\llbracket \alpha \rrbracket_{\alpha} &\triangleq \square \\
NJ\llbracket \tilde{\mu}x.c \rrbracket_{\alpha} &\triangleq \mathbf{let}\, x = \square \,\mathbf{in}\, NJ\llbracket c \rrbracket_{\alpha} \\
NJ\left\llbracket \mathsf{O}^{\vec{B}}[\vec{v}, e] \right\rrbracket_{\alpha} &\triangleq NJ\llbracket e \rrbracket_{\alpha}[\square \;\; \mathsf{O}[\vec{B}, \overrightarrow{NJ\llbracket v \rrbracket}]] \\
NJ\left\llbracket \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y:l}}(\vec{x}).c}\right] \right\rrbracket_{\alpha} &\triangleq \mathbf{case}\, \square \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow NJ\llbracket c \rrbracket_{\alpha}}
\end{aligned}$$

FIGURE 9.14. Translations between λlet and single-consequence $\mu\tilde{\mu}$.

may convert a term of type Int to a term of type $\neg\neg\,\mathsf{Int}$, whereas here a term of type Int would be converted to another term of type Int.

Theorem 9.1 ($LJ\llbracket\_\rrbracket$ preserves types)**.** *For any λlet derivations of $\Theta \vdash_{\mathcal{G}} C : \mathcal{T}$ and $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A$, there is a single-consequence $\mu\tilde{\mu}$ derivation of $\Gamma \vdash^{\Theta}_{\mathcal{G}} LJ\llbracket v\rrbracket : A \mid$.*

Proof. By induction on the given λlet derivation. For example, the translation of the *Let* rule is:

$$\dfrac{\dfrac{\dfrac{\overset{\vdots\; IH}{\Gamma \vdash^{\Theta}_{\mathcal{G}} LJ\llbracket v\rrbracket : A \mid} \quad \Theta \vdash_{\mathcal{G}} A : \mathcal{S} \qquad \dfrac{\dfrac{\overset{\vdots\; IH}{\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} LJ\llbracket v'\rrbracket : C \mid} \quad \Theta \vdash_{\mathcal{G}} C : \mathcal{T} \quad \dfrac{}{\Theta \mid \alpha : C : \alpha \vdash_{\mathcal{G}} C}\,\mathit{VL}}{\langle LJ\llbracket v'\rrbracket \| \alpha\rangle : \left(\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} \alpha : C\right)}\,\mathit{Cut}}{\Gamma \mid \tilde{\mu}x.\, \langle LJ\llbracket v'\rrbracket \| \alpha\rangle : A \vdash^{\Theta}_{\mathcal{G}} \alpha : C}\,\mathit{AL}}{\langle LJ\llbracket v\rrbracket \| \tilde{\mu}x.\, \langle LJ\llbracket v'\rrbracket \| \alpha\rangle\rangle : \left(\Gamma, \Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : C\right)}\,\mathit{Cut}}{\langle LJ\llbracket v\rrbracket \| \tilde{\mu}x.\, \langle LJ\llbracket v'\rrbracket \| \alpha\rangle\rangle : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : C\right)}\,\mathit{CL}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\alpha.\, \langle LJ\llbracket v\rrbracket \| \tilde{\mu}x.\, \langle LJ\llbracket v'\rrbracket \| \alpha\rangle\rangle : C \mid}\,\mathit{AR}$$

Where the additional premise that $\Theta \vdash_{\mathcal{G}} C : \mathcal{T}$ is satisfied by assumption. The other cases follow similarly. □

Theorem 9.2 ($NJ\llbracket\_\rrbracket$ preserves types)**.**

a) For any single-consequence $\mu\tilde{\mu}$ derivation of $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid$, there is a λlet derivation of $\Gamma \vdash^{\Theta}_{\mathcal{G}} NJ\llbracket v\rrbracket : A$.

b) For any single-consequence $\mu\tilde{\mu}$ derivation of $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A\right)$, there is a λlet derivation of $\Gamma \vdash^{\Theta}_{\mathcal{G}} NJ\llbracket c\rrbracket_{\alpha} : A$.

c) For any single-consequence $\mu\tilde{\mu}$ derivation of $\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \alpha : B$, λlet derivation of $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A$, and derivation of $\Theta \vdash_{\mathcal{G}} A : \mathcal{S}$, there is λlet derivation of $\Gamma \vdash^{\Theta}_{\mathcal{G}} NJ\llbracket e\rrbracket_{\alpha}[v] : B$.

Proof. By mutual induction on the given $\mu\tilde{\mu}$ derivations. For example, the AL rule is translated as

$$\dfrac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \quad \Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \overset{\vdots\; IH}{\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} NJ\llbracket c\rrbracket_{\alpha} : B}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{let}\, x = v\, \mathbf{in}\, NJ\llbracket c\rrbracket_{\alpha} : A}\,\mathit{Let}$$

The rest of the cases are similar. □

In terms of the dynamic semantics, the two translations form an *equational correspondence* (Sabry & Felleisen, 1992) between the two languages. Recall from Chapter V that an equational correspondence between λlet and $\mu\tilde{\mu}$ means that the equations between expressions in the two languages are preserved by both translations, and the translations are inverses of one another up to the respective equational theories. This gives us a one-for-one correspondence between the two languages—two terms are equal in λlet if and only if their translation is equal in $\mu\tilde{\mu}$, and so on. However, the correspondence does not hold for all instances of the parametric equational theories; only certain pairs of substitution strategies correspond to one another. More specifically, besides the fact that a call-by-value strategy cannot correspond to a call-by-name one, the correspondence holds for similar strategies which are *strongly focalizing*. The idea of focalization can be translated to natural style as follows.

Definition 9.1 (Focalizing strategy)**.** A λlet substitution strategy $\mathcal{T}$ is *focalizing* if and only if

- variables are values and the empty context is a co-value (as assumed to hold for all strategies),
- data structures built from values and co-data objects are values (i.e. $\mathsf{K}(\vec{A}, \vec{V})$ and $\lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{X{:}k}, \vec{x}] \Rightarrow v}\right\}$ are values), and
- case analysis on data and co-data observations built from values are co-values (i.e. $\mathbf{case}\,\square\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{X{:}k}, \vec{x}) \Rightarrow v}$ and $\square\ \ \mathsf{O}[\vec{A}, \vec{V}]$ are co-values).

But it turns out this still rather loose criteria of focalization isn't enough. We need to impose a stronger focalization criteria which makes sure that the focusing ς rules can always make progress to a (co-)value, and in natural deduction, co-value can be decomposed into smaller co-values.

Definition 9.2 (Strong focalization)**.** A $\mu\tilde{\mu}$ substitution strategy $\mathcal{S}$ is *strongly focalizing* if and only if every (co-)term is either a (co-)value or a $\succ_{\varsigma_{\mathcal{S}}}$ redex.

A λlet substitution strategy $\mathcal{T}$ is *strongly focalizing* if and only if every term is either a value or a $\succ_{\varsigma_{\mathcal{T}}}$ redex, and if $E[F]$ is a co-value then so is F.

Note how unlike terms, the correspondence involving co-terms is rather implicit, since the natural deduction based λlet has no syntactic notion of co-terms, only the meta-syntactic notion of contexts. With the idea of strong focalization, we establish the correspondence between co-terms in the $\mu\tilde{\mu}$-calculus and frame contexts in the λlet-calculus. That is to say, going from sequent calculus to natural deduction, co-terms translate into frame contexts.

Lemma 9.1 (Framing co-terms)**.** *For all e in the single-consequence $\mu\tilde{\mu}$-calculus, if $\alpha \in FV(e)$ then $NJ[\![e]\!]_\alpha \in FrameCxt$.*

Proof. By induction on the syntax of co-terms. The base cases for co-variables, input abstractions, and case abstractions are immediate, while the case for co-data structures follows from the inductive hypothesis and the fact that *FrameCxt*s compose. □

Whereas going from natural deduction to the sequent calculus, co-value contexts are factored out and revealed as co-value co-terms, but only for strongly focalizing λlet substitution strategies. Note that $\ggcurly_R$ denotes the reflexive-transitive (but not compatible) closure of the rewrite relation $\succ_R$.

Lemma 9.2 (Factoring co-values)**.** *For all strongly focalizing λlet-calculus substitution strategies $\mathcal{T}$, $\lambda let_{\mathcal{T}}$-calculus co-values E, terms v, and co-variables α, $\langle LJ[\![E[v]]\!] \| E' \rangle \ggcurly_{\mu_{LJ[\![\mathcal{T}]\!]}} \langle LJ[\![v]\!] \| LJ[\![E]\!]_\alpha \{E'/\alpha\} \rangle$ for all $\mu\tilde{\mu}_{LJ[\![\mathcal{T}]\!]}$-calculus co-values E'.*

Proof. By induction on the syntax of $CoValue_{\mathcal{T}}$ as a subset of *FrameCxt*:

- $\Box$: $\langle LJ[\![v']\!] \| E' \rangle = \langle LJ[\![v']\!] \| \alpha \{E'/\alpha\} \rangle = \langle LJ[\![v']\!] \| LJ[\![\Box]\!]_\alpha \{E'/\alpha\} \rangle$
- $\mathbf{let}\, x = E \,\mathbf{in}\, v$:

$$\begin{aligned} \langle LJ[\![\mathbf{let}\, x = E[v'] \,\mathbf{in}\, v]\!] \| E' \rangle &\succ_{\mu_{LJ[\![\mathcal{T}]\!]}} \langle LJ[\![E[v']]\!] \| \tilde{\mu}x.\, \langle LJ[\![v]\!] \| E' \rangle \rangle \\ &\ggcurly_{IH} \langle LJ[\![v']\!] \| LJ[\![E]\!]_\alpha \{\tilde{\mu}x.\, \langle LJ[\![v]\!] \| E' \rangle / \alpha\} \rangle \\ &= \langle LJ[\![v']\!] \| LJ[\![\mathbf{let}\, x = E \,\mathbf{in}\, v]\!]_\alpha \{E'/\alpha\} \rangle \end{aligned}$$

which follows since $\mathbf{let}\, x = \Box \,\mathbf{in}\, v \in CoValue_{\mathcal{T}}$ because $\mathcal{T}$ is strongly focalizing.

- $\mathbf{case}\, E\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}$:

$$
\begin{aligned}
&\left\langle LJ \left\llbracket \mathbf{case}\, E[v']\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} \right\rrbracket \middle\| E' \right\rangle \\
&\succ_{\mu_{LJ\llbracket\mathcal{T}\rrbracket}} \left\langle LJ\llbracket E[v'] \rrbracket \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{x}).\langle LJ\llbracket v\rrbracket \| E'\rangle}\right] \right\rangle \\
&\ggcurly_{IH} \left\langle LJ\llbracket v' \rrbracket \middle\| LJ\llbracket E\rrbracket_{\alpha} \left\{\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{x}).\langle LJ\llbracket v\rrbracket \| E'\rangle}\right]/\alpha\right\} \right\rangle \\
&= \left\langle LJ\llbracket v' \rrbracket \middle\| LJ \left\llbracket \mathbf{case}\, E\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} \right\rrbracket_{\alpha} \{E'/\alpha\} \right\rangle
\end{aligned}
$$

which follows since $\mathbf{case}\, \square\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}\, \mathbf{in}\, v \in \mathit{CoValue}_{\mathcal{T}}$ because $\mathcal{T}$ is strongly focalizing.

- $E\ \ \mathsf{O}[\vec{B}, \vec{v}]$:

$$
\begin{aligned}
\left\langle LJ \left\llbracket E[v']\ \ \mathsf{O}[\vec{B}, \vec{v}] \right\rrbracket \middle\| E' \right\rangle &\succ_{\mu_{LJ\llbracket\mathcal{T}\rrbracket}} \left\langle LJ\llbracket E[v'] \rrbracket \middle\| \mathsf{O}^{\vec{B}}[\overrightarrow{LJ\llbracket v\rrbracket}, E'] \right\rangle \\
&\ggcurly_{IH} \left\langle LJ\llbracket v' \rrbracket \middle\| LJ\llbracket E\rrbracket_{\alpha} \left\{\mathsf{O}^{\vec{B}}[\overrightarrow{LJ\llbracket v\rrbracket}, E']/\alpha\right\} \right\rangle \\
&= \left\langle LJ\llbracket v' \rrbracket \middle\| LJ \left\llbracket E\ \ \mathsf{O}[\vec{B}, \vec{v}] \right\rrbracket_{\alpha} \{E'/\alpha\} \right\rangle
\end{aligned}
$$

which follows since $\square\ \ \mathsf{O}[\vec{B}, \vec{v}] \in \mathit{CoValue}_{\mathcal{T}}$ because $\mathcal{T}$ is strongly focalizing.

□

By making the correspondence between the different forms of co-values in the two languages more concrete, we can demonstrate the first part of the equational correspondence that the two language's rewriting rules are sound under translation with respect to each other.

Lemma 9.3 (Soundness of $LJ\llbracket\_\rrbracket$)**.** *For any strongly focalizing λlet substitution strategy $\mathcal{T}$ and pure declarations $\mathcal{G}$, if $v \succ_R v'$ then $LJ\llbracket v\rrbracket\ LJ\llbracket \succ_R\rrbracket\ LJ\llbracket v'\rrbracket$ where the translation of the rewrite relation $\succ_R$ is defined by cases on R as:*

$$
\begin{aligned}
&LJ\llbracket \succ_{let_{\mathcal{T}}}\rrbracket \triangleq \to_{\tilde{\mu}_{LJ\llbracket\mathcal{T}\rrbracket}} \to_{\eta_\mu}
&& LJ\llbracket \succ_{\eta_{let\mathcal{T}}}\rrbracket \triangleq =_{\eta_{\tilde{\mu}}\mu_{\mathcal{S}}\eta_\mu}
&& LJ\llbracket \succ_{cc_{\mathcal{T}}}\rrbracket \triangleq =_{\mu_{LJ\llbracket\mathcal{T}\rrbracket}\eta_\mu} \\
&LJ\left\llbracket \succ_{\beta_{\mathcal{T}}}\right\rrbracket \triangleq \to_{\beta_{LJ\llbracket\mathcal{T}\rrbracket}} \to_{\eta_\mu}
&& LJ\left\llbracket \succ_{\nu^{\forall}{<}\mathsf{Ord}}\right\rrbracket \triangleq \succ_{\nu^{\forall}{<}\mathsf{Ord}} \\
&LJ\left\llbracket \succ_{\beta^{\mathsf{F}}}\right\rrbracket \triangleq \to_{\beta^{\mathsf{F}}} \twoheadleftarrow_{\mu_{LJ\llbracket\mathcal{T}\rrbracket}}
&& LJ\left\llbracket \prec_{\eta^{\mathsf{F}}}\right\rrbracket \triangleq \twoheadleftarrow_{\eta^{\mathsf{F}}\eta_\mu\mu_{LJ\llbracket\mathcal{T}\rrbracket}}
\end{aligned}
$$

Proof. Note that substitution commutes with translation (i.e. $LJ[\![v]\!]\,\{LJ[\![V]\!]/x\} = LJ[\![v\,\{V/x\}]\!]$ and $LJ[\![v]\!]\,\{A/X\} = LJ[\![v\,\{A/X\}]\!]$). Soundness follows by cases on the rewrite rule R, where we let $\mathcal{S} = LJ[\![\mathcal{T}]\!]$.

- $let_{\mathcal{T}}$:

$$\begin{aligned}
LJ[\![\mathbf{let}\,x = V\,\mathbf{in}\,v]\!] &= \mu\alpha.\,\langle LJ[\![V]\!]\|\tilde{\mu}x.\,\langle LJ[\![v]\!]\|\alpha\rangle\rangle \\
&\to_{\tilde{\mu}_{\mathcal{S}}} \mu\alpha.\,\langle LJ[\![v]\!]\,\{LJ[\![V]\!]/x\}\|\alpha\rangle \\
&\to_{\eta_\mu} LJ[\![v]\!]\,\{LJ[\![V]\!]/x\} \\
&= LJ[\![v\,\{V/x\}]\!]
\end{aligned}$$

- $\eta_{let\mathcal{T}}$:

$$\begin{aligned}
LJ[\![\mathbf{let}\,x = v\,\mathbf{in}\,E[x]]\!] &= \mu\alpha.\,\langle LJ[\![v]\!]\|\tilde{\mu}x.\,\langle LJ[\![E[x]]\!]\|\alpha\rangle\rangle \\
&\twoheadrightarrow_{\mu_{\mathcal{S}}} \mu\alpha.\,\langle LJ[\![v]\!]\|\tilde{\mu}x.\,\langle x\|LJ[\![E]\!]_\alpha\rangle\rangle && (\textit{Lemma } 9.2) \\
&\to_{\eta_{\tilde{\mu}}} \mu\alpha.\,\langle LJ[\![v]\!]\|LJ[\![E]\!]_\alpha\rangle \\
&\twoheadleftarrow_{\mu_{\mathcal{S}}} \mu\alpha.\,\langle LJ[\![E[v]]\!]\|\alpha\rangle && (\textit{Lemma } 9.2) \\
&\to_{\eta_\mu} LJ[\![E[v]]\!]
\end{aligned}$$

- $cc_{\mathcal{T}}$:

$$\begin{aligned}
&LJ[\![E[\mathbf{let}\,x = v'\,\mathbf{in}\,v]]\!] \\
&\leftarrow_{\eta_\mu} \mu\alpha.\,\langle LJ[\![E[\mathbf{let}\,x = v'\,\mathbf{in}\,v]]\!]\|\alpha\rangle \\
&\twoheadrightarrow_{\mu_{\mathcal{S}}} \mu\alpha.\,\langle LJ[\![\mathbf{let}\,x = v'\,\mathbf{in}\,v]\!]\|LJ[\![E]\!]_\alpha\rangle && (\textit{Lemma } 9.2) \\
&= \mu\alpha.\,\langle \mu\beta.\,\langle LJ[\![v']\!]\|\tilde{\mu}x.\,\langle LJ[\![v]\!]\|\beta\rangle\rangle\|LJ[\![E]\!]_\alpha\rangle \\
&\to_{\mu_{\mathcal{S}}} \mu\alpha.\,\langle LJ[\![v']\!]\|\tilde{\mu}x.\,\langle LJ[\![v]\!]\|LJ[\![E]\!]_\alpha\rangle\rangle \\
&\twoheadleftarrow_{\mu_{\mathcal{S}}} \mu\alpha.\,\langle LJ[\![v']\!]\|\tilde{\mu}x.\,\langle LJ[\![E[v]]\!]\|\alpha\rangle\rangle && (\textit{Lemma } 9.2) \\
&= LJ[\![\mathbf{let}\,x = v'\,\mathbf{in}\,E[v]]\!]
\end{aligned}$$

- $\beta_{\mathcal{T}}$ for data types:

$$LJ\left[\!\!\left[\mathbf{case}\ \mathsf{K}(\vec{B},\vec{V})\,\mathbf{of}\,\overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l},\,\vec{x}) \Rightarrow v_i}^{i}\right]\!\!\right]$$

$$
\begin{aligned}
&= \mu\alpha.\left\langle \mathsf{K}_i^{\vec{B}}(\overrightarrow{LJ\llbracket V\rrbracket}) \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{x}).\langle LJ\llbracket v_i\rrbracket\|\alpha\rangle}^{i}\right]\right\rangle \\
&\to_{\beta_{\mathcal{S}}} \mu\alpha.\left\langle LJ\llbracket v_i\rrbracket\left\{\overrightarrow{B/Y}, \overrightarrow{LJ\llbracket V\rrbracket/x}\right\}\middle\|\alpha\right\rangle \\
&\to_{\eta_\mu} LJ\llbracket v_i\rrbracket\left\{\overrightarrow{B/Y}, \overrightarrow{LJ\llbracket V\rrbracket/x}\right\} \\
&= LJ\left\llbracket v_i\left\{\overrightarrow{B/Y}, \overrightarrow{V/x}\right\}\right\rrbracket
\end{aligned}
$$

– $\beta_{\mathcal{T}}$ for co-data types:

$$
\begin{aligned}
&LJ\left\llbracket \lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v_i}^{i}\right\}\ \mathsf{O}_i[\vec{B}, \vec{V}]\right\rrbracket \\
&= \mu\alpha.\left\langle \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y{:}l}}[\vec{x}, \alpha].\langle LJ\llbracket v_i\rrbracket\|\alpha\rangle}^{i}\right) \middle\| \mathsf{O}_i^{\vec{B}}[\overrightarrow{LJ\llbracket V\rrbracket}, \alpha]\right\rangle \\
&\to_{\beta_{\mathcal{S}}} \mu\alpha.\left\langle LJ\llbracket v_i\rrbracket\left\{\overrightarrow{B/Y}, \overrightarrow{LJ\llbracket V\rrbracket/x}\right\}\middle\|\alpha\right\rangle \\
&\to_{\eta_\mu} LJ\llbracket v_i\rrbracket\left\{\overrightarrow{B/Y}, \overrightarrow{LJ\llbracket V\rrbracket/x}\right\} \\
&= LJ\left\llbracket v_i\left\{\overrightarrow{B/Y}, \overrightarrow{V/x}\right\}\right\rrbracket
\end{aligned}
$$

– $\nu[\forall_{<\mathsf{Ord}}]$:

$$
\begin{aligned}
&LJ\llbracket \lambda\{x \mathbin{@} j{<}N \Rightarrow v\}\rrbracket \\
&= \mu([j{<}N \mathbin{@_x} \alpha].\langle LJ\llbracket v\rrbracket\|\alpha\rangle) \\
&\succ_{\nu^{\forall_{<\mathsf{Ord}}}} \mu([i{<}N \mathbin{@} \alpha].\langle LJ\llbracket v\rrbracket\,\{\mu([j{<}i \mathbin{@_x} \alpha].\langle LJ\llbracket v\rrbracket\|\alpha\rangle)/x\}\|\alpha\rangle) \\
&= LJ\llbracket \lambda\{\ @\ i{<}N \Rightarrow v\,\{\lambda\{x \mathbin{@} j{<}i \Rightarrow v\}/x\}\}\rrbracket
\end{aligned}
$$

– β^{F} for a data type F:

$$
\begin{aligned}
&LJ\left\llbracket \mathbf{case}\ \mathsf{K}(\vec{B}, \vec{v'})\ \mathbf{of}\ \overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v_i}^{i}\right\rrbracket \\
&= \mu\alpha.\left\langle \mathsf{K}_i^{\vec{B}}(\overrightarrow{LJ\llbracket v'\rrbracket}) \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{x}).\langle LJ\llbracket v_i\rrbracket\|\alpha\rangle}^{i}\right]\right\rangle \\
&\to_{\beta^{\mathsf{F}}} \mu\alpha.\left\langle \overrightarrow{LJ\llbracket v'\rrbracket}\middle\|\tilde{\mu}\vec{x}.\left\langle LJ\llbracket v_i\rrbracket\left\{\overrightarrow{B/Y}\right\}\middle\|\alpha\right\rangle\right\rangle \\
&\twoheadleftarrow_{\mu_{\mathcal{S}}} LJ\left\llbracket \mathbf{let}\ \overrightarrow{x = v'}\ \mathbf{in}\ v_i\left\{\overrightarrow{B/Y}\right\}\right\rrbracket
\end{aligned}
$$

– β^{G} for a co-data type G:

$$
\begin{aligned}
&LJ\left\llbracket \lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v_i}^{i}\right\}\ \mathsf{O}_i[\vec{B}, \vec{v'}]\right\rrbracket \\
&= \mu\alpha.\left\langle \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y:l}}[\vec{x}, \alpha].\langle LJ\llbracket v_i\rrbracket \| \alpha\rangle}^{i}\right)\middle\| \mathsf{O}_i^{\vec{B}}[\overrightarrow{LJ\llbracket v'\rrbracket}, \alpha]\right\rangle \\
&\to_{\beta^{\mathsf{G}}} \mu\alpha.\left\langle \overrightarrow{LJ\llbracket v'\rrbracket}\middle\|\tilde{\mu}\vec{x}.\left\langle LJ\llbracket v_i\rrbracket\left\{\overrightarrow{B/Y}\right\}\middle\|\alpha\right\rangle\right\rangle \\
&\twoheadleftarrow_{\mu_{\mathcal{S}}} LJ\left\llbracket \mathbf{let}\,\overrightarrow{x = v'}\,\mathbf{in}\,v_i\left\{\overrightarrow{B/Y}\right\}\right\rrbracket
\end{aligned}
$$

– η^{F} for a data type F:

$$
\begin{aligned}
&LJ\left\llbracket \mathbf{case}\,v\,\mathbf{of}\,\overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow \mathsf{K}_i(\overrightarrow{Y{:}l}, \vec{x})}^{i}\right\rrbracket \\
&= \mu\alpha.\left\langle LJ\llbracket v\rrbracket\middle\|\tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y:l}}(\vec{x}).\left\langle \mathsf{K}_i^{\overrightarrow{Y:l}}(\vec{x})\middle\|\alpha\right\rangle}^{i}\right]\right\rangle \\
&\to_{\eta^{\mathsf{G}}} \mu\alpha.\langle LJ\llbracket v\rrbracket\|\alpha\rangle \\
&\to_{\eta_\mu} LJ\llbracket v\rrbracket
\end{aligned}
$$

– η^{G} for a co-data type G:

$$
\begin{aligned}
&LJ\left\llbracket \lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow y\ \ \mathsf{O}_i[\overrightarrow{Y{:}l}, \vec{x}]}^{i}\right\}\right\rrbracket \\
&= \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y:l}}[\vec{x}, \alpha].\left\langle \mu\alpha.\left\langle y\middle\|\mathsf{O}_i^{\overrightarrow{Y:l}}[\vec{x}, \alpha]\right\rangle\middle\|\alpha\right\rangle}^{i}\right) \\
&\to_{\mu_{\mathcal{S}}} \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y:l}}[\vec{x}, \alpha].\left\langle y\middle\|\mathsf{O}_i^{\overrightarrow{Y:l}}[\vec{x}, \alpha]\right\rangle}^{i}\right) \\
&\to_{\eta^{\mathsf{G}}} y = LJ\llbracket y\rrbracket
\end{aligned}
$$

□

Lemma 9.4. *For any $\mathcal{S}$ and $\mathcal{T} = NJ\llbracket\mathcal{S}\rrbracket$, and any c and e in the single-consequence $\mu\tilde{\mu}$-calculus,*

a) $NJ\llbracket E\rrbracket_\alpha[NJ\llbracket c\rrbracket_\beta] \twoheadrightarrow_{cc_{\mathcal{T}}} NJ\llbracket c\{E/\beta\}\rrbracket_\alpha$, and

b) $NJ\llbracket E\rrbracket_\alpha[NJ\llbracket e\rrbracket_\beta[NJ\llbracket v\rrbracket]] \twoheadrightarrow_{cc_{\mathcal{T}}} NJ\llbracket e\{E/\beta\}\rrbracket_\alpha[v]$.

Proof. By mutual induction on the syntax of commands and co-terms:

$$\begin{aligned}
NJ[\![E]\!]_\alpha[NJ[\![\langle v\|e\rangle]\!]_\beta] &= NJ[\![E]\!]_\alpha[NJ[\![e]\!]_\beta[NJ[\![v]\!]] \\
&\twoheadrightarrow_{IH} NJ[\![e\,\{E/\beta\}]\!]_\alpha[NJ[\![v]\!]] \\
&= NJ[\![\langle v\|e\rangle\,\{E/\beta\}]\!]_\alpha
\end{aligned}$$

$$NJ[\![E]\!]_\alpha[NJ[\![\beta]\!]_\beta[v']] = NJ[\![E]\!]_\alpha[v']$$

$$\begin{aligned}
NJ[\![E]\!]_\alpha[NJ[\![\tilde{\mu}x.c]\!]_\beta[v']] &= NJ[\![E]\!]_\alpha[\mathbf{let}\,x = v'\,\mathbf{in}\,NJ[\![c]\!]_\beta] \\
&\to_{cc_{\mathcal{T}}} \mathbf{let}\,x = v'\,\mathbf{in}\,NJ[\![E]\!]_\alpha[NJ[\![c]\!]_\beta] \\
&\twoheadrightarrow_{IH} \mathbf{let}\,x = v'\,\mathbf{in}\,NJ[\![c\,\{E/\beta\}]\!]_\alpha \\
&= NJ[\![\tilde{\mu}x.c\,\{E/\beta\}]\!]_\alpha[v']
\end{aligned}$$

$$\begin{aligned}
NJ[\![E]\!]_\alpha\left[NJ\left[\!\!\left[\tilde{\mu}\left[\overrightarrow{K^{\overrightarrow{Y{:}l}}(\vec{x}).c}\right]\right]\!\!\right]_\beta[v']\right] &= NJ[\![E]\!]_\alpha\left[\mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l},\vec{x}) \Rightarrow NJ[\![c]\!]_\beta}\right] \\
&\to_{cc_{\mathcal{T}}} \mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l},\vec{x}) \Rightarrow NJ[\![E]\!]_\alpha[NJ[\![c]\!]_\beta]} \\
&\twoheadrightarrow_{IH} \mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l},\vec{x}) \Rightarrow NJ[\![c\,\{E/\beta\}]\!]_\alpha} \\
&= NJ\left[\!\!\left[\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y:l}(\vec{x}).c\,\{E/\beta\}}\right]\right]\!\!\right]_\alpha[v']
\end{aligned}$$

$$\begin{aligned}
NJ[\![E]\!]_\alpha\left[NJ\left[\!\!\left[\mathsf{O}^{\vec{B}}[\vec{v},e]\right]\!\!\right]_\beta[v']\right] &= NJ[\![E]\!]_\alpha\left[NJ[\![e]\!]_\beta\left[v'\ \ \mathsf{O}[\vec{B},\overrightarrow{NJ[\![v]\!]}]\right]\right] \\
&\twoheadrightarrow_{IH} NJ[\![e\,\{E/\beta\}]\!]_\alpha\left[v'\ \ \mathsf{O}[\vec{B},\overrightarrow{NJ[\![v]\!]}]\right] \\
&= NJ\left[\!\!\left[\mathsf{O}^{\vec{B}}[\vec{v},e\,\{E/\beta\}]\right]\!\!\right]_\alpha[v'] \qquad \Box
\end{aligned}$$

Lemma 9.5 (Soundness of $NJ[\![\_]\!]$)**.** *For any strongly focalizing substitution strategy $\mathcal{S}$ and pure declarations $\mathcal{G}$ in the single-consequence $\mu\tilde{\mu}$-calculus,*

a) if v is ς-normal and $v \succ_R v'$ then $NJ[\![v]\!]\ NJ[\![\succ_R]\!]\ NJ[\![v']\!]$,

b) if c is ς-normal and $c \succ_R c'$ then $NJ[\![c]\!]_\alpha\ NJ[\![\succ_R]\!]\ NJ[\![c']\!]_\alpha$,

c) if e is ς-normal and $e \succ_R e'$ then $NJ[\![e]\!]_\alpha[v]\ NJ[\![\succ_R]\!]\ NJ[\![e']\!]_\alpha[v]$ for all v, *where the translation of the rewrite relation $\succ_R$ is defined by cases on R as:*

$$\begin{aligned}
NJ[\![\succ_{\mu_{\mathcal{S}}}]\!] &\triangleq \twoheadrightarrow_{cc_{NJ[\![\mathcal{S}]\!]}} & NJ[\![\succ_{\tilde{\mu}_{\mathcal{S}}}]\!] &\triangleq \succ_{let_{NJ[\![\mathcal{S}]\!]}} \\
NJ\left[\!\!\left[\succ_{\eta_\mu}\right]\!\!\right] &\triangleq =_\alpha & NJ\left[\!\!\left[\succ_{\eta_{\tilde{\mu}}}\right]\!\!\right] &\triangleq \succ_{\eta_{letNJ[\![\mathcal{S}]\!]}\,let_{NJ[\![\mathcal{S}]\!]}} \\
NJ\left[\!\!\left[\succ_{\beta_{\mathcal{S}}}\right]\!\!\right] &\triangleq \to_{\beta_{NJ[\![\mathcal{S}]\!]}} \twoheadrightarrow_{cc_{NJ[\![\mathcal{S}]\!]}} & NJ\left[\!\!\left[\succ_{\nu^{\forall<\mathsf{Ord}}}\right]\!\!\right] &\triangleq \succ_{\nu^{\forall<\mathsf{Ord}}} \\
NJ\left[\!\!\left[\succ_{\beta^{\mathsf{F}}}\right]\!\!\right] &\triangleq \to_{\beta^{\mathsf{F}}} & NJ\left[\!\!\left[\prec_{\eta^{\mathsf{F}}}\right]\!\!\right] &\triangleq \prec_{\eta^{\mathsf{F}}}
\end{aligned}$$

Proof. Note that $\mathcal{S}$-values are translated to $NJ[\![\mathcal{S}]\!]$-values and substitution for value and type variables commutes with translation (i.e. $NJ[\![v]\!]\,\{NJ[\![V]\!]/x\} = NJ[\![v\,\{V/x\}]\!]$, etc.). Soundness follows by cases on the rewrite rule R, where we let $\mathcal{T} = NJ[\![\mathcal{S}]\!]$.

– $\mu_{\mathcal{S}}$:

$$NJ[\![\langle\mu\beta.c\|E\rangle]\!]_\alpha = NJ[\![E]\!]_\alpha[NJ[\![c]\!]_\beta] \twoheadrightarrow_{cc_{\mathcal{T}}} NJ[\![c\,\{E/\beta\}]\!]_\alpha \qquad (\textit{Lemma } 9.4)$$

– $\tilde{\mu}_{\mathcal{S}}$:

$$\begin{aligned}
NJ[\![\langle V\|\tilde{\mu}x.c\rangle]\!]_\alpha &= \mathbf{let}\, x = NJ[\![V]\!]\,\mathbf{in}\, NJ[\![c]\!]_\alpha \\
&\succ_{let_{\mathcal{T}}} NJ[\![c]\!]_\alpha\,\{NJ[\![V]\!]/x\} \\
&= NJ[\![c\,\{V/x\}]\!]_\alpha
\end{aligned}$$

– η_μ: $NJ[\![\mu\alpha.\,\langle v\|\alpha\rangle]\!] = NJ[\![\alpha]\!]_\alpha[NJ[\![v]\!]] = NJ[\![v]\!]$

– $\eta_{\tilde{\mu}}$: by assuming ς-normality, there are only two cases

$$NJ[\![\tilde{\mu}x.\,\langle x\|E\rangle]\!]_\alpha[v] = \mathbf{let}\, x = v\,\mathbf{in}\, NJ[\![E]\!]_\alpha[v] \succ_{\eta_{let\mathcal{T}}} NJ[\![E]\!]_\alpha[v]$$

$$\begin{aligned}
NJ[\![\tilde{\mu}x.\,\langle x\|\tilde{\mu}y.c\rangle]\!]_\alpha[v] &= \mathbf{let}\, x = v\,\mathbf{in}\,\mathbf{let}\, y = x\,\mathbf{in}\, NJ[\![c]\!]_\alpha \\
&\succ_{let_{\mathcal{T}}} \mathbf{let}\, x = v\,\mathbf{in}\, NJ[\![c]\!]_\alpha\,\{x/y\} \\
&=_\alpha \mathbf{let}\, y = v\,\mathbf{in}\, NJ[\![c]\!]_\alpha \\
&= NJ[\![\tilde{\mu}y.c]\!]_\alpha[v]
\end{aligned}$$

- $\beta_{\mathcal{S}}$ for data types:

$$
\begin{aligned}
&NJ\left\llbracket\left\langle \mathsf{K}_i^{\vec{B}}(\vec{V}) \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{x}).c_i}^{\,i}\right]\right\rangle\right\rrbracket_{\alpha} \\
&= \textbf{case } \mathsf{K}_i(\vec{B}, \overrightarrow{NJ\llbracket V\rrbracket}) \textbf{ of } \overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow NJ\llbracket c_i\rrbracket_{\alpha}}^{\,i} \\
&\succ_{\beta_{\mathcal{T}}} NJ\llbracket c_i\rrbracket_{\alpha}\left\{\overrightarrow{B/Y}, NJ\llbracket V\rrbracket/x\right\} \\
&= NJ\left\llbracket c_i\left\{\overrightarrow{B/Y}, V/x\right\}\right\rrbracket_{\alpha}
\end{aligned}
$$

- $\beta_{\mathcal{S}}$ for co-data types:

$$
\begin{aligned}
&NJ\left\llbracket\left\langle \mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y{:}l}}[\vec{x}, \beta].c_i}^{\,i}\right) \middle\| \mathsf{O}_i^{\vec{B}}[\vec{V}, E]\right\rangle\right\rrbracket_{\alpha} \\
&= NJ\llbracket E\rrbracket_{\alpha}\left[\lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow NJ\llbracket c_i\rrbracket_{\beta}}^{\,i}\ \mathsf{O}_i[\vec{B}, \overrightarrow{NJ\llbracket V\rrbracket}]\right\}\right] \\
&\to_{\beta_{\mathcal{T}}} NJ\llbracket E\rrbracket_{\alpha}\left[NJ\llbracket c_i\rrbracket_{\beta}\left\{\overrightarrow{B/Y}, \overrightarrow{NJ\llbracket V\rrbracket/x}\right\}\right] \\
&= NJ\llbracket E\rrbracket_{\alpha}\left[NJ\left\llbracket c_i\left\{\overrightarrow{B/Y}, \overrightarrow{V/x}\right\}\right\rrbracket_{\beta}\right] \\
&\twoheadrightarrow_{cc_{\mathcal{T}}} NJ\left\llbracket c_i\left\{\overrightarrow{B/Y}, \overrightarrow{V/x}, E/\beta\right\}\right\rrbracket_{\alpha} && (\textit{Lemma } 9.4)
\end{aligned}
$$

- $\nu^{\forall_{<\mathsf{Ord}}}$:

$$
\begin{aligned}
NJ\llbracket\mu(j{<}N \mathbin{@_x} \alpha.c)\rrbracket &= \lambda\{x \mathbin{@} j{<}N \Rightarrow NJ\llbracket c\rrbracket_{\alpha}\} \\
&\succ_{\nu^{\forall_{<\mathsf{Ord}}}} \lambda\{\ @\ i{<}N \Rightarrow NJ\llbracket c\rrbracket_{\alpha}\{\lambda\{x \mathbin{@} j{<}i \Rightarrow NJ\llbracket c\rrbracket_{\alpha}\}/x\}\} \\
&= NJ\llbracket\mu(i{<}N \mathbin{@} \alpha.c\{\mu(j{<}i \mathbin{@_x} \alpha.c)/x\})\rrbracket
\end{aligned}
$$

- β^{F} for a data type F:

$$
\begin{aligned}
&NJ\left\llbracket\left\langle \mathsf{K}_i^{\vec{B}}(\vec{v}) \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{x}).c_i}^{\,i}\right]\right\rangle\right\rrbracket_{\alpha} \\
&= \textbf{case } \mathsf{K}_i(\vec{B}, \overrightarrow{NJ\llbracket v\rrbracket}) \textbf{ of } \overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow NJ\llbracket c_i\rrbracket_{\alpha}}^{\,i} \\
&\succ_{\beta^{\mathsf{F}}} \textbf{let } \overrightarrow{x = NJ\llbracket v\rrbracket} \textbf{ in } NJ\llbracket c_i\rrbracket_{\alpha}\left\{\overrightarrow{B/Y}\right\} \\
&= NJ\left\llbracket\left\langle \vec{v} \middle\| \tilde{\mu}\vec{x}.c_i\left\{\overrightarrow{B/Y}\right\}\right\rangle\right\rrbracket_{\alpha}
\end{aligned}
$$

– β^{G} for a co-data type G:

$$\begin{aligned}
&NJ\left\llbracket\left\langle\mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y{:}l}}[\vec{x},\beta].c_i}^{i}\right)\middle\|\mathsf{O}_i^{\vec{B}}[\vec{v},e]\right\rangle\right\rrbracket_{\alpha}\\
&= NJ\llbracket e\rrbracket_{\alpha}\left[\lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l},\vec{x}] \Rightarrow NJ\llbracket c_i\rrbracket_{\beta}}^{i}\ \ \mathsf{O}_i[\vec{B},\overrightarrow{NJ\llbracket v\rrbracket}]\right\}\right]\\
&\to_{\beta^{\mathsf{G}}} NJ\llbracket e\rrbracket_{\alpha}\left[\mathbf{let}\,\overrightarrow{x = NJ\llbracket v\rrbracket}\,\mathbf{in}\,NJ\llbracket c_i\rrbracket_{\beta}\left\{\overrightarrow{B/Y}\right\}\right]\\
&= NJ\left\llbracket\left\langle\mu\beta.\left\langle\vec{v}\middle\|\tilde{\mu}\vec{x}.c_i\left\{\overrightarrow{B/Y}\right\}\right\rangle\middle\|e\right\rangle\right\rrbracket_{\alpha}
\end{aligned}$$

– η^{F} for a data type F:

$$\begin{aligned}
NJ\left\llbracket\tilde{\mu}\left[\overrightarrow{\mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{x}).\left\langle\mathsf{K}_i^{\overrightarrow{Y{:}l}}(\vec{x})\middle\|\alpha\right\rangle}^{i}\right]\right\rrbracket_{\alpha}[v] &= \mathbf{case}\,v\,\mathbf{of}\,\overrightarrow{\mathsf{K}_i(\overrightarrow{Y{:}l},\vec{x}) \Rightarrow \mathsf{K}_i(\overrightarrow{Y{:}l},\vec{x})}^{i}\\
&\succ_{\eta^{\mathsf{F}}} v = NJ\llbracket\alpha\rrbracket_{\alpha}[v]
\end{aligned}$$

– η^{G} for a co-data type G:

$$\begin{aligned}
NJ\left\llbracket\mu\left(\overrightarrow{\mathsf{O}_i^{\overrightarrow{Y{:}l}}[\vec{x},\alpha].\left\langle y\middle\|\mathsf{O}_i^{\overrightarrow{Y{:}l}}[\vec{x},\alpha]\right\rangle}^{i}\right)\right\rrbracket &= \lambda\left\{\overrightarrow{\mathsf{O}_i[\overrightarrow{Y{:}l},\vec{x}] \Rightarrow y\ \ \mathsf{O}_i[\overrightarrow{Y{:}l},\vec{x}]}^{i}\right\}\\
&\succ_{\eta^{\mathsf{G}}} y = NJ\llbracket y\rrbracket \qquad \square
\end{aligned}$$

The second part of the equational correspondence is to demonstrate that the two translations are inverses of one another, meaning that every round-trip translation is equal to the starting point. This round-trip equality does not depend on the focalization properties of substitution strategies, but instead assumes that we work with pre-focused, ς-normal expressions. The reason for this assumption is to midigate the subtle difference between co-terms like $v \cdot e$ in $\mu\tilde{\mu}$ and frame contexts like $F[\square \cdot v]$ in λlet. In particular, due to the ordering imposed by the $\mu\tilde{\mu}$ ς rules, the term v has the first priority in $v \cdot e$ and e is lifted out and evaluated second. In λlet, however, the context F always has implicit priority $F[\square \cdot v]$ if F is not an evaluation context, and v only gets discovered and evaluated when it lands in the eye of an evaluation context. By ς-normalizing beforehand, this difference in discovery and evaluation is eliminated entirely.

Lemma 9.6 (Natural deduction round trip)**.** *For any $\mathcal{T}$, if v is ς-normal then $NJ\llbracket LJ\llbracket v\rrbracket\rrbracket = v$.*

Proof. By induction on the syntax of terms.

- x: $NJ\llbracket LJ\llbracket x\rrbracket\rrbracket = x$

- $\mathbf{let}\, x = v' \,\mathbf{in}\, v$:

$$\begin{aligned} NJ\llbracket LJ\llbracket \mathbf{let}\, x = v' \,\mathbf{in}\, v\rrbracket\rrbracket &= NJ\llbracket \mu\alpha.\, \langle LJ\llbracket v'\rrbracket \| \tilde{\mu}x.\, \langle LJ\llbracket v\rrbracket \| \alpha\rangle\rangle\rrbracket \\ &= \mathbf{let}\, x = NJ\llbracket LJ\llbracket v'\rrbracket\rrbracket \,\mathbf{in}\, NJ\llbracket LJ\llbracket v\rrbracket\rrbracket \\ &=_{IH} \mathbf{let}\, x = v' \,\mathbf{in}\, v \end{aligned}$$

- $\mathsf{K}(\vec{B},\vec{V})$: $NJ\Big\llbracket LJ\Big\llbracket \mathsf{K}(\vec{B},\vec{V})\Big\rrbracket\Big\rrbracket = \mathsf{K}(\vec{B}, \overrightarrow{NJ\llbracket LJ\llbracket V\rrbracket\rrbracket}) =_{IH} \mathsf{K}(\vec{B},\vec{V})$

- $\mathbf{case}\, v' \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}$:

$$\begin{aligned} & NJ\Big\llbracket LJ\Big\llbracket \mathbf{case}\, v' \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}\Big\rrbracket\Big\rrbracket \\ &= NJ\Big\llbracket \mu\alpha.\, \Big\langle LJ\llbracket v'\rrbracket \Big\| \tilde{\mu}\Big[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{x}).\langle LJ\llbracket v\rrbracket \| \alpha\rangle}\Big]\Big\rangle\Big\rrbracket \\ &= \mathbf{case}\, NJ\llbracket LJ\llbracket v'\rrbracket\rrbracket \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow NJ\llbracket LJ\llbracket v\rrbracket\rrbracket} \\ &=_{IH} \mathbf{case}\, v' \,\mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} \end{aligned}$$

- $\lambda\Big\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\Big\}$:

$$\begin{aligned} NJ\Big\llbracket LJ\Big\llbracket \lambda\Big\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\Big\}\Big\rrbracket\Big\rrbracket &= NJ\Big\llbracket \mu\Big(\overrightarrow{\mathsf{O}^{\overrightarrow{Y{:}l}}[\vec{x}, \alpha].\langle LJ\llbracket v\rrbracket \| \alpha\rangle}\Big)\Big\rrbracket \\ &= \lambda\Big\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow NJ\llbracket LJ\llbracket v\rrbracket\rrbracket}\Big\} \\ &=_{IH} \lambda\Big\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\Big\} \end{aligned}$$

- $v'\ \ \mathsf{O}[\vec{B},\vec{V}]$:

$$\begin{aligned} NJ\Big\llbracket LJ\Big\llbracket v'\ \ \mathsf{O}[\vec{B},\vec{V}]\Big\rrbracket\Big\rrbracket &= NJ\Big\llbracket \mu\alpha.\, \Big\langle LJ\llbracket v'\rrbracket \Big\| \mathsf{O}^{\vec{B}}[\vec{V}, \alpha]\Big\rangle\Big\rrbracket \\ &= NJ\llbracket LJ\llbracket v'\rrbracket\rrbracket\ \ \mathsf{O}[\vec{B}, \overrightarrow{NJ\llbracket LJ\llbracket V\rrbracket\rrbracket}] \\ &=_{IH} v'\ \ \mathsf{O}[\vec{B},\vec{V}] \end{aligned}$$

□

Lemma 9.7 (Sequent calculus round trip)**.** *For any $\mathcal{S}$ in the single-consequence, higher-order $\mu\tilde{\mu}$-calculus,*

a) if v is ς-normal then $LJ[\![NJ[\![v]\!]]\!] =_{\mu_{\mathcal{S}}\eta_\mu} v$,

b) if c is ς-normal then $LJ[\![NJ[\![c]\!]_\alpha]\!] =_{\mu_{\mathcal{S}}\eta_\mu} \mu\alpha.c$,

c) if e is ς-normal then for all v, $LJ[\![NJ[\![e]\!]_\alpha[v]]\!] =_{\mu_{\mathcal{S}}\eta_\mu} \mu\alpha.\left\langle LJ[\![v]\!]\middle\|e\right\rangle$

Proof. By (mutual) induction on the syntax of commands, terms, and co-terms.

- x: $LJ[\![NJ[\![x]\!]]\!] = x$

- $\mu\alpha.c$: $LJ[\![NJ[\![\mu\alpha.c]\!]]\!] = LJ[\![NJ[\![c]\!]_\alpha]\!] =_{IH} \mu\alpha.c$

- $\mathsf{K}^{\vec{B}}(\vec{V})$: $LJ\left[\!\!\left[NJ\left[\!\!\left[\mathsf{K}^{\vec{B}}(\vec{V})\right]\!\!\right]\right]\!\!\right] = \mathsf{K}^{\vec{B}}(\overrightarrow{LJ[\![NJ[\![V]\!]]\!]}) =_{IH} \mathsf{K}^{\vec{B}}(\vec{V})$

- $\mu\left(\overrightarrow{\mathsf{O}^{Y:l}[\vec{x},\alpha].c}\right)$:

$$\begin{aligned}
LJ\left[\!\!\left[NJ\left[\!\!\left[\mu\left(\overrightarrow{\mathsf{O}^{Y:l}[\vec{x},\alpha].c}\right)\right]\!\!\right]\right]\!\!\right] &= \mu\left(\overrightarrow{\mathsf{O}^{Y:l}[\vec{x},\alpha].\left\langle LJ[\![NJ[\![c]\!]_\alpha]\!]\middle\|\alpha\right\rangle}\right)\\
&=_{IH} \mu\left(\overrightarrow{\mathsf{O}^{Y:l}[\vec{x},\alpha].\langle\mu\alpha.c\|\alpha\rangle}\right)\\
&=_{\mu_{\mathcal{S}}} \mu\left(\overrightarrow{\mathsf{O}^{Y:l}[\vec{x},\alpha].c}\right)
\end{aligned}$$

- $\langle v\|e\rangle$:

$$LJ[\![NJ[\![\langle v\|e\rangle]\!]_\alpha]\!] = LJ[\![NJ[\![e]\!]_\alpha[NJ[\![v]\!]]]\!] =_{IH} \mu\alpha.\left\langle LJ[\![NJ[\![v]\!]]\!]\middle\|e\right\rangle =_{IH} \mu\alpha.\langle v\|e\rangle$$

- α: $LJ[\![NJ[\![\alpha]\!]_\alpha[v]]\!] = LJ[\![v]\!] =_{\eta_\mu} \mu\alpha.\langle LJ[\![v]\!]\|\alpha\rangle$

- $\tilde{\mu}x.c$:

$$\begin{aligned}
LJ[\![NJ[\![\tilde{\mu}x.c]\!]_\alpha[v]]\!] &= LJ[\![\mathbf{let}\, x = v\, \mathbf{in}\, NJ[\![c]\!]_\alpha]\!]\\
&= \mu\alpha.\left\langle LJ[\![v]\!]\middle\|\tilde{\mu}x.\left\langle LJ[\![NJ[\![c]\!]_\alpha]\!]\middle\|\alpha\right\rangle\right\rangle\\
&=_{IH} \mu\alpha.\langle LJ[\![v]\!]\|\tilde{\mu}x.\langle\mu\alpha.c\|\alpha\rangle\rangle\\
&=_{\mu_{\mathcal{S}}} \mu\alpha.\langle LJ[\![v]\!]\|\tilde{\mu}x.c\rangle
\end{aligned}$$

- $\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y:l}}(\vec{x}).c}\right]$:

$$\begin{aligned}
LJ\left\llbracket NJ\left\llbracket \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y:l}}(\vec{x}).c}\right]\right\rrbracket_{\alpha}[v]\right\rrbracket &= LJ\left\llbracket \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y:l}, \vec{x}) \Rightarrow NJ\llbracket c\rrbracket_{\alpha}}\right\rrbracket \\
&= \mu\alpha.\left\langle LJ\llbracket v\rrbracket \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y:l}(\vec{x}).\left\langle LJ\llbracket NJ\llbracket c\rrbracket_{\alpha}\rrbracket \middle\| \alpha\right\rangle}\right]\right\rangle \\
&=_{IH} \mu\alpha.\left\langle LJ\llbracket v\rrbracket \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y:l}(\vec{x}).\langle \mu\alpha.c \| \alpha\rangle}\right]\right\rangle \\
&=_{\mu_{\mathcal{S}}} \mu\alpha.\left\langle LJ\llbracket v\rrbracket \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y:l}(\vec{x}).c}\right]\right\rangle
\end{aligned}$$

- $\mathsf{O}^{\vec{B}}[\vec{V}, E]$:

$$\begin{aligned}
LJ\left\llbracket NJ\left\llbracket \mathsf{O}^{\vec{B}}[\vec{V}, E]\right\rrbracket_{\alpha}[v]\right\rrbracket &= LJ\left\llbracket NJ\llbracket E\rrbracket_{\alpha}[v\ \ \mathsf{O}[\vec{B}, \overrightarrow{NJ\llbracket V\rrbracket}]]\right\rrbracket \\
&=_{IH} \mu\alpha.\left\langle LJ\left\llbracket v\ \ \mathsf{O}[\vec{B}, \overrightarrow{NJ\llbracket V\rrbracket}]\right\rrbracket \middle\| E\right\rangle \\
&= \mu\alpha.\left\langle \mu\alpha.\left\langle LJ\llbracket v\rrbracket \middle\| \mathsf{O}^{\vec{B}}[\overrightarrow{LJ\llbracket NJ\llbracket V\rrbracket\rrbracket}, \alpha]\right\rangle \middle\| E\right\rangle \\
&=_{IH} \mu\alpha.\left\langle \mu\alpha.\left\langle LJ\llbracket v\rrbracket \middle\| \mathsf{O}^{\vec{B}}[\vec{V}, \alpha]\right\rangle \middle\| E\right\rangle \\
&=_{\mu_{\mathcal{S}}} \mu\alpha.\left\langle LJ\llbracket v\rrbracket \middle\| \mathsf{O}^{\vec{B}}[\vec{V}, E]\right\rangle \qquad \square
\end{aligned}$$

With the soundness of equations and the round-trip equalities established for ς-normal forms, we already have an equational correspondence between the ς-normal sub-languages of λlet and $\mu\tilde{\mu}$. This correspondence can be generalized to the full languages by noting that ς-normalization commutes with all other rewriting rules.

Theorem 9.3 (Parametric equational correspondence)**.** *For any stable substitution strategies $\mathcal{S}$ and $\mathcal{T} = NJ\llbracket\mathcal{S}\rrbracket$, the $\lambda let_{\mathcal{T}}$-calculus is in equational correspondence with the single-consequence $\mu\tilde{\mu}_{\mathcal{S}}$-calculus.*

Proof. First, note that there is an equational correspondence between the ς-normal sub-languages of the two calculi. That the equations are sound with respect to translation of ς-normal forms follows from Lemmas 9.3 and 9.5 since the translations are compositional, so compatibility holds automatically (Downen & Ariola, 2014a). That the translations are inverses on ς-normal forms up to the equational theory is shown in Lemmas 9.6 and 9.7.

Additionally, note that ς reduction is strongly normalizing (because each reduction strictly decreases the number of non-(co-)values in a data structure or observation), so every expression has a ς-normal form. Also note that because of stability, ς-normalization commutes with all other reductions: if $v \twoheadrightarrow_{\varsigma} v_1 \not\rightarrow_{\varsigma}$ and $v \rightarrow_R v_2$ then $v_1 \twoheadrightarrow_R v'$ and $v_2 \twoheadrightarrow_{\varsigma} v'$ for some v', and similar for commands and co-terms. This follow by induction on the syntax of (co-)terms and commands, where the only interest case is when a lifted non-(co-)value reduces to a (co-)value, which is undone by $\mu_{\mathcal{S}}\tilde{\mu}_{\mathcal{S}}\eta_{\mu}\eta_{\tilde{\mu}}$ in the $\mu\tilde{\mu}$-calculus (as a generalization of Lemma 1 in Johnson-Freyd *et al.* (2017)) and $let_{\mathcal{T}}\eta_{let}$ in $\lambda let_{\mathcal{T}}$. For example, we could have the divergent reductions

$$\begin{aligned}&\iota_1\,(x) \twoheadrightarrow_{\varsigma} \mathbf{let}\,x = v'\,\mathbf{in}\,\iota_1\,(x) \not\rightarrow_{\varsigma}\\ &\iota_1\,(v) \rightarrow_{\beta} \iota_1\,(V)\end{aligned}$$

which is brought back together from the inductive hypothesis $v' \twoheadrightarrow V' \twoheadleftarrow V$ as follows:

$$\mathbf{let}\,x = v\,\mathbf{in}\,\iota_1\,(x) \twoheadrightarrow \mathbf{let}\,x = V'\,\mathbf{in}\,\iota_1\,(x) \rightarrow_{let_{\mathcal{T}}} \iota_1\,(V') \twoheadleftarrow_{\varsigma} \iota_1\,(V)$$

Therefore, ς-normalization also forms an equational correspondence between the full languages and the ς-normal sub-languages, and thus the full $\mu\tilde{\mu}_{\mathcal{S}}$ and $\lambda let_{\mathcal{T}}$ calculi are in equational correspondence, because equational correspondences compose (Sabry & Felleisen, 1992). □

As a result of this static and dynamic correspondence, we can transfer results from the sequent calculus to natural deduction. In particular, many of the applications of orthogonality models from Chapter VII can be directly used to prove similar properties about λlet. For example, the logic of λlet is consistent by composing Theorems 9.1 and 7.1, so that there's no well-typed closed term of an arbitrary type variable, i.e. $\vdash_{\mathcal{G}}^{X:\mathcal{S}} v : X$ is not derivable, since that would imply a contradictory well-typed closed command in $\mu\tilde{\mu}$ by instantiating X with A and $\neg A$ for example. Note that this is the same as saying there is no well-typed closed term of the empty data type $\vdash_{\mathcal{G}} v : 0$, due to the strength of the $0E$ elimination rule corresponding to the *ex falso quodlibet* principle: "from false, anything follows." We also get strong normalization of the (weak or bounded) $let_{\mathcal{T}}\beta_{\mathcal{T}}\nu^{\forall<\mathsf{Ord}}$ reduction theory for any strongly focalizing substitution strategy $\mathcal{T}$ by composing Lemma 9.3 and Theorem 7.3 because they translate to at least one reduction in $\mu\tilde{\mu}$, which can be extended to include $\varsigma_{\mathcal{T}}$ reductions by noting that they are strongly normalizing on their own and commute with the other reductions as in the proof of Theorem 9.3 so they can't introduce an infinite reduction sequence.

Additionally, we know that the entire λlet equational theory, including the η law for (co-)data types, is coherent for the polarized $\mathcal{P} = \mathcal{V}, \mathcal{N}$ strategy since $\iota_1\,() \neq \iota_2\,()$ in a closed environment as that would cause a contradiction with Theorems 9.3 and 7.5.

Remark 9.2. Note that, while here we apply the results from orthogonality models of the sequent calculus to a natural deduction language via translation, we could also build an orthogonality model for the natural deduction λlet-calculus directly. The way that we have phrased the definitions (of computational poles, interaction spaces, orthogonality, safety conditions, worlds, and types) in Chapter VII is quite general, and can be applied to other families of languages, too. For example, we could build a model directly the call-by-name instance of the λlet-calculus by setting the $\top\!\!\top$ pole in the assumed safety condition to be the set of all λ-calculus terms so that the computation relation is the reduction of terms. The untyped space $\mathbb{U} = (\mathbb{U}_+, \mathbb{U}_-)$ of the world can be defined so that the positive side $\mathbb{U}_+$ is the set of all λlet-calculus terms and the negative side $\mathbb{U}_-$ is the set of all λlet-calculus frame contexts. That way, the cut operation can be defined as context filling: given $v, F \in \mathbb{U}$, then $\langle v \| F \rangle = F[v]$. Furthermore, the value space $\mathbb{V}$ of the world is the set of terms (because all terms are values in $\mathcal{N}$) and the set of call-by-name evaluation contexts (which are $\mathcal{N}$ co-values).

In this setup, we can construct the model for functions as the negatively-constructed type built around applicative contexts:

$$\mathbb{A} \to_- \mathbb{B} = \mathit{Neg}(\{E[\Box\ v] \mid v \in \mathbb{A}, E \in \mathbb{B}\})$$

As it turns out, this negative definition of function types in terms of orthogonality and applicative contexts is logically the same as the more traditional (unary) logical relation model for functions:

$$\mathbb{A} \to_{LR} \mathbb{B} = \{v \in \mathit{Term} \mid \forall v' \in \mathbb{A}, v\ v' \in \mathbb{B}\}$$

In particular, suppose $v \in \mathbb{A} \to_- \mathbb{B}$ for some semantic types $\mathbb{A}$ and $\mathbb{B}$, so the fact that $\langle v \| E[\Box\ v] \rangle \in \perp\!\!\!\perp$ implies that $\langle v\ v' \| E \rangle \in \perp\!\!\!\perp$ because cutting is context filling in the λ-calculus: $\langle v \| E[\Box\ v'] \rangle = E[v\ v'] = \langle v\ v' \| E \rangle$. That means $v\ v' \in \mathbb{B}$ since $\mathbb{B} = \mathbb{B}^{\perp\!\!\!\perp_{\mathbb{W}}}$, which in turn means that $v \in \mathbb{A} \to_{LR} \mathbb{B}$. Going the other way, suppose $v \in \mathbb{A} \to_{LR} \mathbb{B}$, so the fact that $v\ v' \in \mathbb{B}$ for all $v' \in \mathbb{A}$ (which comes by definition) implies that $\langle v\ v' \| E \rangle \in \perp\!\!\!\perp$ since $\mathbb{B}$ is a $\perp\!\!\!\perp$-space. As before, we have $\langle v\ v' \| E \rangle = E[v\ v'] = \langle v \| E[\Box\ v'] \rangle \in \perp\!\!\!\perp$, meaning that $v \in \mathbb{A} \to_- \mathbb{B}$ by Lemma 7.6. *End remark* 9.2.

$$v \in \mathit{Term} ::= \dots \mid \mu\alpha.c \qquad\qquad c \in \mathit{Command} ::= \langle v \| \alpha \rangle$$

$$\mathit{Judgement} ::= (\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta) \mid c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$$

Control rules:

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \alpha : A, \Delta}{\langle v \| \alpha \rangle : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)}\ \mathit{Pass} \qquad \frac{c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\alpha.c : A \mid \Delta}\ \mathit{Act}$$

Control kinding rules:

$$\frac{\Gamma \vdash_{\mathcal{G}} v :: \mathcal{R} \mid \alpha :: \mathcal{R}, \Delta}{\langle v \| \alpha \rangle :: \left(\Gamma \vdash_{\mathcal{G}} \alpha :: \mathcal{R}, \Delta\right)}\ \mathit{Pass} \qquad \frac{c :: \left(\Gamma \vdash_{\mathcal{G}} \alpha :: \mathcal{R}, \Delta\right)}{\Gamma \vdash_{\mathcal{G}} \mu\alpha.c :: \mathcal{R} \mid \Delta}\ \mathit{Act}$$

FIGURE 9.15. $\lambda\mu$: adding multiple consequences to natural deduction.

Multiple Consequences

The natural deduction λ*let*-calculus corresponds to the $\mu\tilde{\mu}$ sequent calculus restricted to a single consequence. What, then, is the natural deduction equivalent to the entire $\mu\tilde{\mu}$-calculus? Since multiple consequences in the classical sequent calculus let us write additional programs, we need some way to represent multiple consequences in natural deduction as well. As it turns out, Parigot's (1992) $\lambda\mu$-calculus for representing classical logic in natural deduction style gives us exactly what we need to extend the pure λ-calculus with co-variables representing other side consequences besides the main return value of the term.

The extension of the pure λ*let*-calculus with co-variables is the parametric $\lambda\mu$*let*-calculus whose additional syntax and typing rules are shown in Figure 9.15. The $\lambda\mu$*let*-calculus extend λ*let* with commands of the form $\langle v \| \alpha \rangle$ and μ-abstractions of the form $\mu\alpha.c$. Note that commands and μ-abstractions are similar to constructs of the same name in the sequent calculus, except that a command is always between a term and a co-variable: $\lambda\mu$*let* still has no notion of co-terms which represent the left rules exclusive to the sequent calculus. The two new typing rules are for activation (*Act*) that forms a term by abstracting over a co-variable in a command (and thus activating its output as the output of the term), and passivation (*Pass*) that forms a command by throwing the output of a term to some free co-variable in scope.

$$
\begin{array}{llll}
(\mu_{\mathcal{T}}) & E[\mu\alpha.c] \succ_{\mu_{\mathcal{T}}} \mu\beta.c\,\{\langle E\|\beta\rangle/\langle\Box\|\alpha\rangle\} & & (\Box \neq E \in \mathit{CoValue}_{\mathcal{T}} \\
 & & & \;\beta \notin FV(E)) \\
(\mu_\alpha) & \langle\mu\alpha.c\|\beta\rangle \succ_{\mu_\alpha} c\,\{\beta/\alpha\} & & \\
(\eta_\mu) & \mu\alpha.\,\langle v\|\alpha\rangle \succ_{\eta_\mu} v & & (\alpha \notin FV(v)) \\
(cc_\mu) & \mu\alpha.\,\langle \mathbf{let}\, x = v' \,\mathbf{in}\, v\|\beta\rangle \succ_{cc_\mu} \mathbf{let}\, x = v' \,\mathbf{in}\, \mu\alpha.\,\langle v\|\beta\rangle & & (\alpha \notin FV(v')) \\
(cc_\mu) & \mu\alpha.\left\langle \begin{array}{c}\mathbf{case}\, v' \,\mathbf{of} \\ \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}\end{array} \middle\| \beta \right\rangle \succ_{cc_\mu} \begin{array}{c}\mathbf{case}\, v' \,\mathbf{of} \\ \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow \mu\alpha.\,\langle v\|\beta\rangle}\end{array} & & (\alpha \notin \bigcup\{\overrightarrow{FV(v')}\})
\end{array}
$$

FIGURE 9.16. The laws of control in $\lambda\mu$.

The $\lambda\mu let$ also extend λlet's dynamic semantics with the additional rewriting rules in rewriting rules shown in Figure 9.16, which are also parameterized by the same concept of substitution strategy as λlet. The $\mu_{\mathcal{T}}$ law incrementally captures co-value contexts, representing a portion of the full co-value, by performing a *structural substitution* (Ariola & Herbelin, 2008) of command-delimited co-values for commands containing co-variables. The substitution $\{\langle E\|\beta\rangle/\langle\Box\|\alpha\rangle\}$ means to follow the usual capture-avoiding substitution rules to replace every command of the form $\langle v\|\alpha\rangle$, where v can be any term, with $\langle E[v']\|\alpha\rangle$ where $v' = \{\langle E\|\beta\rangle/\langle\Box\|\alpha\rangle\}$ is the structural substitution applied to that sub-term v. The μ_α law finishes off capturing a co-value by substituting one co-variable for another. The η_μ law is analogous to the η_μ law in the sequent calculus, which eliminates a redundant μ-abstraction. And finally, we have the cc_μ laws for commuting conversions of control, which are analogous to the $cc_{\mathcal{T}}$ commuting conversions which push co-values into **let** and **case** expressions. Since the term $\mu\alpha.\,\langle v\|\beta\rangle$ means to swap co-values, naming the surrounding co-value α and instead use the one named β for evaluating v, it is subject to the same control flow as a co-value.

Sharing, control, and join points

Compilers must balance many conflicting concerns when optimizing programs. For example, there is often a tradeoff between space and time which impacts the way in which programs can be effectively transformed. In particular, in compilers that use term-based representations like the λ-calculus, this tradeoff can be most easily seen in reductions that duplicate terms. The most common form of duplication is caused by substitution used by β and other laws, which repeats the same full value, whose description may be very large, for every occurence of the substituted variable. Because of duplication, "simpler" terms do not always result in "better" code.

Consider the following term, where the locally bound variable f is referenced three times to form the three components of the resulting triple and where the term v in the body of f is something very large.

$$\begin{aligned}
&\mathbf{let}\, f = \lambda x.\, \mathbf{case}\, \mathit{even}\ x\, \mathbf{of} \\
&\qquad\qquad\qquad \mathsf{True} \Rightarrow v \\
&\qquad\qquad\qquad \mathsf{False} \Rightarrow \mathsf{Nothing} \\
&\mathbf{in}\ \ (f, f\ z, f\ 1)
\end{aligned}$$

A straightforward *let* reduction would replicate the abstraction $\lambda x.\, \mathbf{case}\, \mathit{even}\ x\, \mathbf{of}\ \dots$ three times to place it in each context where f is used. Often, putting a value in its context can open up new opportunities for simplifications or optimizations which lead to better code, but not always. In this example, the first use of f is as a component in a data structure (the triple), and so substituting the value for f opens up no new opportunities for simplification whatsoever. The second use of f is provided the argument z, so substituting a value for this f does open up a new β reduction, but the term gets immediately stuck on the free variable z after, so this, too, is not helpful. The second use of f, which is provided the concrete argument 1, is very different to the first two cases, however. After substituting the value for f, a β reduction plugs in 1 for its parameter x, and the body of the function can then be simplified down to the simple result $\mathsf{Nothing}$. Therefore, in the context of compiling programs, it is often better to selectively *inline* values for only specific references to variables that might lead to useful further simplifications. In the above example, we would only want to inline a value for the third occurence of f, to get the term

$$\begin{aligned}
&\mathbf{let}\, f = \lambda x.\, \mathbf{case}\, \mathit{even}\ x\, \mathbf{of} \\
&\qquad\qquad\qquad \mathsf{True} \Rightarrow v \\
&\qquad\qquad\qquad \mathsf{False} \Rightarrow \mathsf{Nothing} \\
&\mathbf{in}\ \ (f, f\ z, \mathsf{Nothing})
\end{aligned}$$

which simplifies the known call to f without unhelpfully duplicating the large sub-term v.

Substitution of values for variables is not the only cause for replication and increase of code size, however. For example, consider the following λlet-term which corresponds to expressions that routinely arise when simplifying functional programs.

$$\begin{array}{l}
\mathbf{case}\left(\begin{array}{l}\mathbf{case}\, z\, \mathbf{of} \\ \quad \begin{array}{ll} \mathsf{A}(x,y) & \Rightarrow y \\ \mathsf{B}(x) & \Rightarrow \mathsf{Just}(x) \\ \mathsf{C} & \Rightarrow v_c \end{array}\end{array}\right)\mathbf{of} \\
\quad \begin{array}{ll} \mathsf{Just}(x) & \Rightarrow v_1 \\ \mathsf{Nothing} & \Rightarrow v_0 \end{array}
\end{array}$$

In this term, we have a **case** that is discriminating the result of another **case** expression. The thing to notice is that, in the case that z is $\mathsf{B}(x)$, the inner **case** returns $\mathsf{Just}(x)$ to the outer **case**, which can then be simplified to v_0. It would be better to make this observation about information and control flow explicit in the term and to shortcut the production and consumption of the intermediate short-lived value $\mathsf{Just}(x)$ which creates extra garbage only to directly marshall a result from one **case** to another. But the way that the two **case**s are nested keeps $\mathsf{Just}(x)$ away from its surrounding use. But this is exactly what the commuting conversions are good for! A **case** context is a co-value in any focalizing strategy $\mathcal{T}$, so $cc_{\mathcal{T}}$ pushes the outer **case** into the branches of the inner one, opening up the opportunity to simplify the middle B branch as follows:

$$\begin{array}{l}
\mathbf{case}\left(\begin{array}{ll}\mathbf{case}\, z\, \mathbf{of} & \begin{array}{ll} \mathsf{A}(x,y) & \Rightarrow y \\ \mathsf{B}(x) & \Rightarrow \mathsf{Just}(x) \\ \mathsf{C} & \Rightarrow z' \end{array}\end{array}\right)\mathbf{of} \\
\quad \begin{array}{ll} \mathsf{Just}(x) & \Rightarrow v \\ \mathsf{Nothing} & \Rightarrow 0 \end{array}
\end{array}$$

$$
\begin{array}{lll}
\to_{cc_{\mathcal{T}}}\ \mathbf{case}\, z\, \mathbf{of}\ \mathsf{A}(x,y) & \Rightarrow & \mathbf{case}\, y\, \mathbf{of}\ \begin{array}[t]{ll}\mathsf{Just}(x) & \Rightarrow v\\ \mathsf{Nothing} & \Rightarrow 0\end{array}\\
\qquad\qquad\qquad\qquad \mathsf{B}(x) & \Rightarrow & \mathbf{case}\ \mathsf{Just}(x)\, \mathbf{of}\ \begin{array}[t]{ll}\mathsf{Just}(x) & \Rightarrow v\\ \mathsf{Nothing} & \Rightarrow 0\end{array}\\
\qquad\qquad\qquad\qquad \mathsf{C} & \Rightarrow & \mathbf{case}\, z'\, \mathbf{of}\ \begin{array}[t]{ll}\mathsf{Just}(x) & \Rightarrow v\\ \mathsf{Nothing} & \Rightarrow 0\end{array}
\end{array}
$$

$$
\begin{array}{lll}
\to_{\beta_{\mathcal{T}}}\ \mathbf{case}\, z\, \mathbf{of}\ \mathsf{A}(x,y) & \Rightarrow & \mathbf{case}\, y\, \mathbf{of}\ \begin{array}[t]{ll}\mathsf{Just}(x) & \Rightarrow v\\ \mathsf{Nothing} & \Rightarrow 0\end{array}\\
\qquad\qquad\qquad\qquad \mathsf{B}(x) & \Rightarrow & v\\
\qquad\qquad\qquad\qquad \mathsf{C} & \Rightarrow & \mathbf{case}\, z'\, \mathbf{of}\ \begin{array}[t]{ll}\mathsf{Just}(x) & \Rightarrow v\\ \mathsf{Nothing} & \Rightarrow 0\end{array}
\end{array}
$$

The $cc_{\mathcal{T}}$ commuting conversion has allowed us to simplify the B branch and eliminate an unnecessary intermediate value. However, notice how the $cc_{\mathcal{T}}$ commuting conversion has also unfortunately duplicated the full outer **case** into both the A and B branches to no avail, since the **case** is just stuck on the variables y and z'. But worse, the subterm v, which could be very large in practice, has also been inadvertently duplicated. This is not just a mild inconvenience, but actually represents a fatal problem to the use of unrestrained commuting conversions for compilation; unfortunate nestings of **case** can create a chain reaction where commuting conversions increase the size of the program exponentially thereby making compilation unfeasible Lindley (2005).

Lets instead change our viewpoint and look at the problem from the perspective of the sequent calculus. The above nested **case** s in λlet translates to a $\mu\tilde{\mu}$ term of the following form:

$$
\mu\alpha.\left\langle \mu\beta.\left\langle z \middle\| \tilde{\mu}\begin{bmatrix}\mathsf{A}(x,y). & \langle y\|\beta\rangle\\ \mathsf{B}(x) & .\langle \mathsf{Just}(x)\|\beta\rangle\\ \mathsf{C} & .\langle z'\|\beta\rangle\end{bmatrix}\right\rangle \middle\| \tilde{\mu}\begin{bmatrix}\mathsf{Just}(x) & .\langle v\|\alpha\rangle\\ \mathsf{Nothing}. & \langle 0\|\alpha\rangle\end{bmatrix}\right\rangle
$$

Note the key difference here, the outer **case** has been given a name via the co-variable β which is explicitly referenced in the inner **case** . Because of the symmetry and duality of the sequent calculus, we can therefore use exactly the same intuition that we had for ordinary variables: only inline co-values for co-variables that appear in contexts that could lead to useful further simplifications. In this example, only inlining into the B branch is helpful to eliminate the intermediate $\mathsf{Just}(x)$, which leads to the following simplification:

$$\mu\alpha.\left\langle \mu\beta.\left\langle z \middle\| \tilde{\mu}\begin{bmatrix} \mathsf{A}(x,y).\,\langle y\|\beta\rangle \\ \mathsf{B}(x) \quad .\,\langle \mathsf{Just}(x)\|\beta\rangle \\ \mathsf{C} \qquad .\,\langle z'\|\beta\rangle \end{bmatrix} \right\rangle \middle\| \tilde{\mu}\begin{bmatrix} \mathsf{Just}(x)\ .\,\langle v\|\alpha\rangle \\ \mathsf{Nothing}.\,\langle 0\|\alpha\rangle \end{bmatrix} \right\rangle$$

$$=_{\mu_{\mathcal{S}}} \mu\alpha.\left\langle \mu\beta.\left\langle z \middle\| \tilde{\mu}\begin{bmatrix} \mathsf{A}(x,y).\,\langle y\|\beta\rangle \\ \mathsf{B}(x) \quad .\left\langle \mathsf{Just}(x) \middle\| \tilde{\mu}\begin{bmatrix} \mathsf{Just}(x)\ .\,\langle v\|\alpha\rangle \\ \mathsf{Nothing}.\,\langle 0\|\alpha\rangle \end{bmatrix} \right\rangle \\ \mathsf{C} \qquad .\,\langle z'\|\beta\rangle \end{bmatrix} \right\rangle \middle\| \tilde{\mu}\begin{bmatrix} \mathsf{Just}(x)\ .\,\langle v\|\alpha\rangle \\ \mathsf{Nothing}.\,\langle 0\|\alpha\rangle \end{bmatrix} \right\rangle$$

$$=_{\beta_{\mathcal{S}}} \mu\alpha.\left\langle \mu\beta.\left\langle z \middle\| \tilde{\mu}\begin{bmatrix} \mathsf{A}(x,y).\,\langle y\|\beta\rangle \\ \mathsf{B}(x) \quad .\,\langle v\|\alpha\rangle \\ \mathsf{C} \qquad .\,\langle z'\|\beta\rangle \end{bmatrix} \right\rangle \middle\| \tilde{\mu}\begin{bmatrix} \mathsf{Just}(x)\ .\,\langle v\|\alpha\rangle \\ \mathsf{Nothing}.\,\langle 0\|\alpha\rangle \end{bmatrix} \right\rangle$$

This use of explicit control flow in this way to tame commuting conversions mimics Kennedy's (2007) analysis and promotion of *continuation-passing style* (CPS) λ-calculi as a favorable representation compared to direct-style λ-calculi, where co-values take the role of the functional representation of continuations as λ-abstractions in CPS. A CPS aficionado could then view the sequent calculus as a foundation for how defunctionalization (Reynolds, 1998) can be strategically employed in a CPS language, where only the continuations for data types like sums and pairs are abstractions, and the continuations for co-data types like functions are concrete structures that more directly correspond to the original structure of functional programs.

However, there is something interesting to note about the selective inlining of co-values for co-variables. Even though we started with a single-consequence $\mu\tilde{\mu}$ term, coming from a pure λlet term, the result of simplification steps outside of the simplistic, single-consequence regime. Note that the case abstraction

$$\tilde{\mu}\left[\begin{array}{ll} \mathsf{A}(x,y). & \langle y \| \beta \rangle \\ \mathsf{B}(x) & . \, \langle v \| \alpha \rangle \\ \mathsf{C} & . \, \langle z' \| \beta \rangle \end{array}\right]$$

has *two* free co-variables, not one, meaning that it no longer lies in the single-consequence subset of $\mu\tilde{\mu}$. This means that if we want to mimic this same sequent calculus simplification in direct natural deduction style, we also have to step outside the pure λlet-calculus. The need to be more explicit about control flow is one possible application of the classical $\lambda\mu let$-calculus, where we can introduce μ-abstractions which let us speak more clearly about the flow of control in a program and skip over uninteresting control paths. For example, the above $\mu\tilde{\mu}$ simplification can be restated in $\lambda\mu let$ as follows:

$$\begin{array}{l} \textbf{case} \left(\begin{array}{lll} \textbf{case}\, z\, \textbf{of} & \mathsf{A}(x,y) & \Rightarrow y \\ & \mathsf{B}(x) & \Rightarrow \mathsf{Just}(x) \\ & \mathsf{C} & \Rightarrow z' \end{array} \right) \textbf{of} \\ \quad \begin{array}{ll} \mathsf{Just}(x) & \Rightarrow v \\ \mathsf{Nothing} & \Rightarrow 0 \end{array} \end{array}$$

$$=_{\mu_{\mathcal{T}}\eta_\mu cc_\mu} \mu\alpha. \left\langle \begin{array}{l} \textbf{case}\, \mu\beta. \left\langle \begin{array}{lll} \textbf{case}\, z\, \textbf{of} & \mathsf{A}(x,y) & \Rightarrow \mu\delta. \, \langle y \| \beta \rangle \\ & \mathsf{B}(x) & \Rightarrow \mu\delta. \, \langle \mathsf{Just}(x) \| \beta \rangle \\ & \mathsf{C} & \Rightarrow \mu\delta. \, \langle z' \| \beta \rangle \end{array} \middle\| \beta \right\rangle \textbf{of} \\ \quad \begin{array}{ll} \mathsf{Just}(x) & \Rightarrow v \\ \mathsf{Nothing} & \Rightarrow 0 \end{array} \end{array} \middle\| \alpha \right\rangle$$

$$=_{\mu_{\mathcal{T}}} \mu\alpha. \left(\begin{array}{l} \mathbf{case}\,\mu\beta. \left(\begin{array}{lll} \mathbf{case}\,z\,\mathbf{of} & \mathsf{A}(x,y) & \Rightarrow \mu\delta.\,\langle y \| \beta\rangle \\ & \mathsf{B}(x) & \Rightarrow \mu\delta. \left(\begin{array}{l} \mathbf{case}\ \mathsf{Just}(x)\,\mathbf{of} \\ \quad \begin{array}{ll} \mathsf{Just}(x) & \Rightarrow v \\ \mathsf{Nothing} & \Rightarrow 0 \end{array} \end{array} \middle\| \alpha \right) \\ & \mathsf{C} & \Rightarrow \mu\delta.\,\langle z' \| \beta\rangle \end{array} \middle\| \beta \right) \mathbf{of} \\ \quad \begin{array}{ll} \mathsf{Just}(x) & \Rightarrow v \\ \mathsf{Nothing} & \Rightarrow 0 \end{array} \end{array} \middle\| \alpha \right)$$

$$=_{\beta_{\mathcal{T}}} \mu\alpha. \left(\begin{array}{l} \mathbf{case}\,\mu\beta. \left(\begin{array}{lll} \mathbf{case}\,z\,\mathbf{of} & \mathsf{A}(x,y) & \Rightarrow \mu\delta.\,\langle y \| \beta\rangle \\ & \mathsf{B}(x) & \Rightarrow \mu\delta.\,\langle v \| \alpha\rangle \\ & \mathsf{C} & \Rightarrow \mu\delta.\,\langle z' \| \beta\rangle \end{array} \middle\| \beta \right) \mathbf{of} \\ \quad \begin{array}{ll} \mathsf{Just}(x) & \Rightarrow v \\ \mathsf{Nothing} & \Rightarrow 0 \end{array} \end{array} \middle\| \alpha \right)$$

The term that we get at the end corresponds to the same term that we got in $\mu\tilde{\mu}$, where the B branch jumps directly to the surrounding α, skipping the second **case** since it was already done in that branch, while the other branches go to the second **case** via β. In this sense, the co-variable α serves as an explicit *join point*, where the branching control induced by case analysis eventually meets back up; not every case flows through β, but no matter the value of z, eventually some result will be output through α. So from the outside, this term appears to be pure—returning the same result regardless of the context in which it is used—even though internally might use impure jumps to skip past the normal control flow for programs.

However, notice how describing this control flow goes against the natural predilections of the $\lambda\mu let$ natural deduction calculus. In natural deduction *everything* is a term that must return some result, forming an implicit call-and-return control flow in the program. But here we must use μ-abstractions that bind an unused co-variable δ to skip past the implied control flow of **case** expressions and insert our own. In the $\mu\tilde{\mu}$ sequent calculus, in contrast, the control flow of case analysis (on either data structures or co-data observations) is explicit, so selective inlining for co-variables is more natural. This is an example of how not all representations are equal, even if they are equivalently expressive. Representing join points as first-class control is more natural in the sequent style (or continuation-passing style, for the same reasons). For imperative programs, *static single assignment* (SSA) (Cytron *et al.*, 1991) represents join points with *ϕ-nodes* that change their value based on the history of run-time control flow that leads to them.

Alternatively, we can come up with a representation of join points that fits more closely with the idioms of pure functional programming, based on either the sequent calculus (Downen *et al.*, 2016) or natural deduction (Maurer *et al.*, 2017). The purely functional view of join points limits the expressive power over control flow by restricting co-variables based on their static scope (so that co-variables cannot be captured in closure, making them unreachable when they go out of scope) and their static types (limiting co-variables to only data types of the form $\exists X_1{:}k_1 \dots \exists X_n{:}k_n.(A_1 \otimes \dots \otimes A_m)$ in the sequent calculus or co-data types of the form $\forall Z.\forall X_1{:}k_1 \dots \forall X_n{:}k_n.(A_1 \to \dots \to A_m \to Z)$ in natural deduction). This alternative view of join points based on the sequent calculus lets us integrate them directly into common intermediate languages based on the λ-calculus that are used for compiling functional programs, for example the core intermediate language for the Glasgow Haskell Compiler,[3] to improve existing optimizations and enable new ones without disrupting the rest of the language. In other words, there are is a space for languages in between λlet and $\lambda\mu let$ for representing purely functional join points that serve as a practical compromise between the expressiveness of the classical sequent calculus and the referential transparency of the intuitionistic sequent calculus.

Natural deduction vs sequent calculus with multiple consequences

Since we have extended the natural deduction calculus to incorporate co-variables, we can extend the correspondence to cover the entire classical sequent calculus. The translations between the two languages are likewise extended for $\lambda\mu let$ as shown in Figure 9.17. The definition of $LK[\![v]\!]$ is the same as $LJ[\![v]\!]$ in every case except for the new μ-abstractions and commands of $\lambda\mu let$. In contrast, the reverse translation $NK[\![\_]\!]$ requires more changes from $NJ[\![\_]\!]$. In particular, co-terms are now translated as command-delimited frame contexts, and not just frame contexts. For example, instead of translating the co-variable α as the empty context, it is now $\langle \Box \| \alpha \rangle$ because there might be a choice between many different co-variables, so we need to spell out which one to not confuse them. As a consequence of this change, the translations of

[3] See `https://ghc.haskell.org/trac/ghc/wiki/SequentCore` for more information.

let and **case** expressions are now also different; they both make use of a dummy co-variable δ, listed explicitly as the extra parameter in $NK[\![\_]\!]_\delta$, which is effectively ignored at run-time but convenient for coercing commands to terms (via $\mu\delta.c$) and terms to commands (via $\langle v \| \delta \rangle$). For example, in the command $\langle \mathbf{let}\, x = v \,\mathbf{in}\, \mu\delta.c \| \delta \rangle$, if δ is not free in c, then any co-value substituted for the outer δ is discarded as in the following reduction sequence:

$$\begin{aligned} \langle \mathbf{let}\, x = v \,\mathbf{in}\, \mu\delta.c \| \delta \rangle &\twoheadrightarrow \langle \mathbf{let}\, x = V \,\mathbf{in}\, \mu\delta.c \| \delta \rangle \\ &\to_{let_{\mathcal{T}}} \langle \mu\delta.c\,\{V/x\} \| \delta \rangle \\ &\to_{\mu_\alpha} c\,\{V/x\} \end{aligned}$$

So the co-value of the dummy δ does not influence program behavior. However, it is extremely useful for translating input and especially case abstractions, where the commands in the branches of the case may all "return" a result to different places denoted by different co-variables, which goes against the more predictable control flow of the λ-calculus syntax.

Despite the apparent differences between the single-consequence $NJ[\![\_]\!]$ and multiple-consequence $NK[\![\_]\!]$ translations, the two still coincide up to the $\mu\tilde{\mu}$ equational theory for single-consequence commands and (co-)terms. In this sense, $NK[\![\_]\!]$ is a conservative extension of the $NJ[\![\_]\!]$ translation.

Lemma 9.8 (Natural purification)**.** *For any single-consequence $\mu\tilde{\mu}$ command c, term v, or co-term e,*

a) $NK[\![v]\!]_\delta =_{\eta_\mu cc_\mu} NJ[\![v]\!]$ where $\beta \notin FV(v)$ for all co-variables β,

b) $NK[\![c]\!]_\delta =_{\eta_\mu cc_\mu} \langle NJ[\![c]\!]_\alpha \| \alpha \rangle$ for some α such that $\beta \notin FV(NJ[\![c]\!]_\alpha)$ for all co-variables β, and

c) $NK[\![e]\!]_\delta[v] =_{\eta_\mu cc_\mu} \langle NJ[\![e]\!]_\alpha[v] \| \alpha \rangle$ for some α and all v such that $\beta \notin FV(NJ[\![e]\!]_\alpha)$ for all co-variables β when $\delta \notin FV(e)$.

Proof. By mutual induction on the syntax of commands, terms, and co-terms. The cases for cuts, (co-)variables, data structures, and co-data observations follow immediately from the inductive hypothesis. The remaining cases are:

– $\mu\alpha.c$: $NK[\![\mu\alpha.c]\!]_\delta = \mu\alpha.NK[\![c]\!]_\delta =_{IH} \mu\alpha.\, \langle NJ[\![c]\!]_\alpha \| \alpha \rangle =_{\eta_\mu} NJ[\![c]\!]_\alpha = NJ[\![\mu\alpha.c]\!]$

Natural deduction to sequent calculus:

$$LK\llbracket \mu\alpha.c \rrbracket \triangleq \mu\alpha.LK\llbracket c \rrbracket \qquad\qquad LK\llbracket \langle v \| \alpha \rangle \rrbracket \triangleq \langle LK\llbracket v \rrbracket \| \alpha \rangle$$

Sequent calculus to natural deduction:

$$NK\llbracket \langle v \| e \rangle \rrbracket_\delta \triangleq NK\llbracket e \rrbracket_\delta [NK\llbracket v \rrbracket_\delta]$$

$$\begin{aligned}
NK\llbracket x \rrbracket_\delta &\triangleq x \\
NK\llbracket \mu\alpha.c \rrbracket_\delta &\triangleq \mu\alpha.NK\llbracket c \rrbracket_\delta \\
NK\left\llbracket \mathsf{K}^{\vec{B}}(\vec{v}) \right\rrbracket_\delta &\triangleq \mathsf{K}(\vec{B}, \overrightarrow{NK\llbracket v \rrbracket_\delta}) \\
NK\left\llbracket \mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y{:}l}}[\vec{x}, \alpha].c}\right) \right\rrbracket_\delta &\triangleq \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow \mu\alpha.NK\llbracket c \rrbracket_\delta}\right\}
\end{aligned}$$

$$\begin{aligned}
NK\llbracket \alpha \rrbracket_\delta &\triangleq \langle \square \| \alpha \rangle \\
NK\llbracket \tilde{\mu}x.c \rrbracket_\delta &\triangleq \langle \mathbf{let}\, x = \square\, \mathbf{in}\, \mu\delta.NK\llbracket c \rrbracket_\delta \| \delta \rangle \\
NK\left\llbracket \mathsf{O}^{\vec{B}}[\vec{v}, e] \right\rrbracket_\delta &\triangleq NK\llbracket e \rrbracket_\delta [\square \;\; \mathsf{O}[\vec{B}, \overrightarrow{NK\llbracket v \rrbracket_\delta}]] \\
NK\left\llbracket \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{x}).c}\right] \right\rrbracket_\delta &\triangleq \left\langle \mathbf{case}\, \square\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow \mu\delta.NK\llbracket c \rrbracket_\delta} \middle\| \delta \right\rangle
\end{aligned}$$

FIGURE 9.17. Translations between natural deduction and the sequent calculus with many consequences.

- $\mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y{:}l}}[\vec{x},\alpha].c}\right)$:

$$\begin{aligned}
NK\left[\!\!\left[\mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y{:}l}}[\vec{x},\alpha].c}\right)\right]\!\!\right]_{\delta} &= \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l},\vec{x}] \Rightarrow \mu\alpha.NK[\![c]\!]_{\delta}}\right\} \\
&=_{IH} \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l},\vec{x}] \Rightarrow \mu\alpha.\left\langle NJ[\![c]\!]_{\alpha}\middle\|\alpha\right\rangle}\right\} \\
&=_{\eta_\mu} \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l},\vec{x}] \Rightarrow NJ[\![c]\!]_{\alpha}}\right\} \\
&= NJ\left[\!\!\left[\mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y{:}l}}[\vec{x},\alpha].c}\right)\right]\!\!\right]
\end{aligned}$$

- $\tilde{\mu}x.c$ where $\alpha \in FV(c)$:

$$\begin{aligned}
NK[\![\tilde{\mu}x.c]\!]_{\delta}[v] &= \langle \mathbf{let}\, x = v\, \mathbf{in}\, \mu\delta.NK[\![c]\!]_{\delta} \| \delta\rangle \\
&=_{IH} \langle \mathbf{let}\, x = v\, \mathbf{in}\, \mu\delta.\left\langle NJ[\![c]\!]_{\alpha}\|\alpha\right\rangle \| \delta\rangle \\
&=_{cc_\mu} \langle \mu\delta.\left\langle \mathbf{let}\, x = v\, \mathbf{in}\, NJ[\![c]\!]_{\alpha}\|\alpha\right\rangle \| \delta\rangle \\
&=_{\mu_\alpha} \langle \mathbf{let}\, x = v\, \mathbf{in}\, NJ[\![c]\!]_{\alpha} \| \alpha\rangle \\
&= \langle NJ[\![\tilde{\mu}x.c]\!]_{\alpha}[v] \| \alpha\rangle
\end{aligned}$$

- $\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{x}).c}\right]$:

$$\begin{aligned}
NK\left[\!\!\left[\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y{:}l}}(\vec{x}).c}\right]\right]\!\!\right]_{\delta}[v] &= \left\langle \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l},\vec{x}) \Rightarrow \mu\delta.NK[\![c]\!]_{\delta}} \middle\| \delta\right\rangle \\
&=_{IH} \left\langle \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l},\vec{x}) \Rightarrow \mu\delta.\left\langle NJ[\![c]\!]_{\alpha}\|\alpha\right\rangle} \middle\| \delta\right\rangle \\
&=_{cc_\mu} \left\langle \mu\delta.\left\langle \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l},\vec{x}) \Rightarrow NJ[\![c]\!]_{\alpha}} \middle\| \alpha\right\rangle \middle\| \delta\right\rangle \\
&=_{\mu_\alpha} \left\langle \mathbf{case}\, v\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l},\vec{x}) \Rightarrow NJ[\![c]\!]_{\alpha}} \middle\| \alpha\right\rangle \\
&= \left\langle NJ\left[\!\!\left[\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y{:}l}(\vec{x}).c}\right]\right]\!\!\right]_{\alpha} \middle\| \alpha\right\rangle \qquad \square
\end{aligned}$$

Now that we can translate the full classical $\mu\tilde{\mu}$ sequent calculus to the natural deduction $\lambda\mu let$-calculus, the two translations still preserve the static and dynamic semantics of the languages. This correspondence follows similar to the single-consequence one, where the types correspond on the nose and there is an equational correspondence between the two languages. The one wrinkle with the preservation of types is that the $NK\llbracket v\rrbracket_{\delta}$ translation introduces δ as an extra free co-variable, whose type can be anything because its co-value is never used, which should be different from any free or bound co-variable in v to make sure that the translation is still well-typed.

Theorem 9.4 ($LK\llbracket\_\rrbracket$ preserves types)**.** *a) For any λμlet derivation* $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$*, there is a* $\mu\tilde{\mu}$ *derivation of* $LK\llbracket c\rrbracket : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$*.*

b) For any λμlet derivations of $\Theta \vdash_{\mathcal{G}} C : \mathcal{T}$ *and* $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta$*, there is a* $\mu\tilde{\mu}$ *derivation of* $\Gamma \vdash^{\Theta}_{\mathcal{G}} LK\llbracket v\rrbracket : A \mid \Delta$*.*

Proof. By induction on the given *λμlet* derivation. The cases for the subset of *λμlet* rules that in *λlet* are the same as Theorem 9.1. The two new cases are for the *Pass* and *Act* rules which translate as follows:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ IH\\ \Gamma \vdash^{\Theta}_{\mathcal{G}} LK\llbracket v\rrbracket : A \mid \alpha : A, \Delta\end{array} \quad \dfrac{}{\mid \beta : A \vdash^{\Theta}_{\mathcal{G}} \beta : A}\,\mathit{VL}}{\langle LK\llbracket v\rrbracket \| \beta\rangle : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \beta : A, \alpha : A, \Delta\right)}\,\mathit{CR}, \mathit{XR}}{\langle LK\llbracket v\rrbracket \| \alpha\rangle : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)}\,\mathit{Cut} \qquad \dfrac{\begin{array}{c}\vdots\ IH\\ LK\llbracket c\rrbracket : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta\right)\end{array}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\alpha.LK\llbracket c\rrbracket : A \mid \Delta}\,\mathit{AR}$$

Where the additional premise that $\Theta \vdash_{\mathcal{G}} C : \mathcal{T}$ is satisfied by assumption. The other cases follow similarly. □

Theorem 9.5 ($NK\llbracket\_\rrbracket$ preserves types)**.** *Given any* $(\delta : C) \notin \Delta$*,*

a) For any $\mu\tilde{\mu}$ *derivation of* $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta$*, if* $\delta \notin BV(v)$ *then there is a λμlet derivation of* $\Gamma \vdash^{\Theta}_{\mathcal{G}} NK\llbracket v\rrbracket_{\delta} : A \mid \delta : C, \Delta$*.*

b) For any $\mu\tilde{\mu}$ *derivation of* $c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta\right)$*, if* $\delta \notin BV(c)$ *then there is a λμlet derivation of* $NK\llbracket c\rrbracket_{\delta} : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \delta : C, \Delta\right)$*.*

c) For any $\mu\tilde{\mu}$ *derivation of* $\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta$*, λμlet derivation of* $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \delta : C, \Delta$*, and derivation of* $\Theta \vdash_{\mathcal{G}} A : \mathcal{S}$*, if* $\delta \notin BV(e)$ *then there is a λμlet derivation of* $NK\llbracket e\rrbracket_{\delta}[v] : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \delta : C, \Delta\right)$*.*

Proof. By mutual induction on the given $\mu\tilde{\mu}$ derivations, similar to Theorem 9.2. For example, the AL rule is now translated as

$$\dfrac{\dfrac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \delta : C, \Delta \quad \Theta \vdash_{\mathcal{G}} A : \mathcal{S} \quad \dfrac{\begin{array}{c}\vdots\; IH \\ NK[\![c]\!] : \left(\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} \delta : C, \Delta\right)\end{array}}{\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} \mu\delta.NK[\![c]\!] : C \mid \delta : C, \Delta}\,Act}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{let}\, x = v\, \mathbf{in}\, \mu\delta.NK[\![c]\!] : C \mid \delta : C, \Delta}\,Let}{\langle \mathbf{let}\, x = v\, \mathbf{in}\, \mu\delta.NK[\![c]\!] \| \delta \rangle : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \delta : C, \Delta\right)}\,Pass$$

Also, the AR rule is translated as:

$$\dfrac{\begin{array}{c}\vdots\; IH \\ c : \left(\Gamma \vdash^{\Theta}_{\mathcal{G}} \delta : B, \alpha : A, \Delta\right)\end{array}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\alpha.NK[\![c]\!]_{\delta} : A \mid \delta : B, \Delta}\,Act$$

where we rely on the side condition that $\delta \notin BV(\mu\alpha.c)$ which implies that $\delta \neq \alpha$. The rest of the cases are similar. □

Lemma 9.9 (Soundness of $LK[\![\_]\!]$)**.** *For any $\mathcal{T}$, if v is ς-normal and $v \succ_R v'$ then $LK[\![v]\!]$ $LK[\![\succ_R]\!]$ $LK[\![v']\!]$ where the translation of the rewrite relation $\succ_R$ is defined by cases on R as:*

$$\begin{aligned} LK[\![\succ_{\mu_{\mathcal{T}}}]\!] &\triangleq =_{\mu_{LK[\![\mathcal{T}]\!]}\eta_\mu} & \qquad LK[\![\succ_{\mu_\alpha}]\!] &\triangleq \succ_{\mu_{LK[\![\mathcal{S}]\!]}} \\ LK\left[\!\!\left[\succ_{\eta_\mu}\right]\!\!\right] &\triangleq \succ_{\eta_\mu} & \qquad LK\left[\!\!\left[\succ_{cc_\mu}\right]\!\!\right] &\triangleq =_{\mu_\alpha} \end{aligned}$$

Proof. The case for η_μ is immediate and μ_α follows from the fact that co-variable substitution commutes with translation (i.e. $LK[\![c]\!]\,\{\beta/\alpha\} = LK[\![c\,\{\beta/\alpha\}]\!]$). The case for $\mu_{\mathcal{T}}$ follows from (the multiple-consequence generalization of) Lemma 9.2:

$$\begin{aligned} LK[\![E[\mu\alpha.c]]\!] &=_{\mu_{LK[\![\mathcal{T}]\!]}\eta_\mu} \mu\alpha.\,\langle \mu\alpha.LK[\![c]\!] \| E_\alpha \rangle \\ &=_{\mu_{LK[\![\mathcal{T}]\!]}\eta_\mu} \mu\alpha.LK[\![c\,\{\langle E\|\alpha\rangle/\langle\square\|\alpha\rangle\}]\!] \\ &= LK[\![\mu\alpha.c\,\{\langle E\|\alpha\rangle/\langle\square\|\alpha\rangle\}]\!] \end{aligned}$$

where the commutation of substitution and translation in second-to-last step follows by induction on the structure of commands and (co-)terms; every step follows immediately from the definition of structural substitution and the inductive hypothesis with the base cases:

$$\begin{aligned} LK[\![\langle v \| \alpha \rangle]\!]\, \{E_\alpha/\alpha\} &= \langle LK[\![v]\!] \| \alpha \rangle \, \{E_\alpha/\alpha\} \\ &= \langle LK[\![v]\!]\, \{E_\alpha/\alpha\} \| E_\alpha \rangle \\ &=_{IH} \langle LK[\![v \, \{\langle E \| \alpha \rangle / \langle \Box \| \alpha \rangle\}]\!] \| E_\alpha \rangle \\ &=_{\mu_{\mathcal{S}} \eta_\mu} \langle LK[\![E[v \, \{\langle E \| \alpha \rangle / \langle \Box \| \alpha \rangle\}]]\!] \| \alpha \rangle \\ &= LK[\![\langle E[v \, \{\langle E \| \alpha \rangle / \langle \Box \| \alpha \rangle\}] \| \alpha \rangle]\!] \end{aligned}$$

Likewise, the case for cc_μ follows from (the multiple-consequence generalization of) Lemma 9.2, where the case for a **let** expression is:

$$\begin{aligned} LK[\![\mu\alpha.\, \langle \mathbf{let}\, x = v' \,\mathbf{in}\, v \| \beta \rangle]\!] &= \mu\alpha.\, \langle \mu\gamma.\, \langle LK[\![v']\!] \| \tilde{\mu}x.\, \langle LK[\![v]\!] \| \gamma \rangle \rangle \| \beta \rangle \\ &=_{\mu_\alpha} \mu\alpha.\, \langle LK[\![v']\!] \| \tilde{\mu}x.\, \langle LK[\![v]\!] \| \beta \rangle \rangle \\ &=_{\alpha} \mu\gamma.\, \langle LK[\![v']\!] \| \tilde{\mu}x.\, \langle LK[\![v]\!]\, \{\gamma/\alpha\} \| \beta \rangle \rangle \\ &=_{\mu_\alpha} \mu\gamma.\, \langle LK[\![v']\!] \| \tilde{\mu}x.\, \langle \mu\alpha.\, \langle LK[\![v]\!] \| \beta \rangle \| \gamma \rangle \rangle \\ &= LK[\![\mathbf{let}\, x = v' \,\mathbf{in}\, \mu\alpha.\, \langle v \| \beta \rangle]\!] \end{aligned}$$

And the case for a **case** expression is similar. □

Lemma 9.10 (Soundness of $NK[\![\_]\!]$)**.** *For any strongly focalizing $\mathcal{S}$ and declarations $\mathcal{G}$,*

a) if v is ς-normal and $v \succ_R v'$ then $NK[\![v]\!]_\delta \succ_R NK[\![v']\!]_\delta$,

b) if c is ς-normal and $c \succ_R c'$ then $NK[\![c]\!]_\delta \succ_R NK[\![c']\!]_\delta$,

c) if e is ς-normal and $e \succ_R e'$ then $NK[\![e]\!]_\delta[v] \succ_R NK[\![e']\!]_\delta[v]$ for all v,

where the translation of the rewrite relation $\succ_R$ is defined by cases on R as:

$$\begin{aligned} NK[\![\succ_{\mu_{\mathcal{S}}}]\!] &\triangleq \rightarrow_{\mu_{NK[\![\mathcal{S}]\!]}} \succ_{\mu_\alpha} & NK[\![\succ_{\tilde{\mu}_{\mathcal{S}}}]\!] &\triangleq \rightarrow_{let_{NK[\![\mathcal{S}]\!]}} \succ_{\mu_\alpha} \\ NK\left[\!\left[\succ_{\eta_\mu}\right]\!\right] &\triangleq \succ_{\eta_\mu} & NK\left[\!\left[\succ_{\eta_{\tilde{\mu}}}\right]\!\right] &\triangleq =_{let_{\mathcal{S}} \eta_{let} \mu_\alpha cc_\mu cc_{NK[\![\mathcal{S}]\!]} \sigma_{NK[\![\mathcal{S}]\!]}} \\ NK\left[\!\left[\succ_{\beta_{\mathcal{S}}}\right]\!\right] &\triangleq \rightarrow_{\beta_{NK[\![\mathcal{S}]\!]}} \twoheadrightarrow_{\mu_{NK[\![\mathcal{S}]\!]}} \rightarrow_{\mu_\alpha} & NK\left[\!\left[\succ_{\nu^{\forall<\mathsf{Ord}}}\right]\!\right] &\triangleq \succ_{\nu^{\forall<\mathsf{Ord}}} \\ NK\left[\!\left[\succ_{\beta^{\mathsf{F}}}\right]\!\right] &\triangleq \rightarrow_{\beta^{\mathsf{F}}} =_{\mu_\alpha cc_\mu} & NK\left[\!\left[\prec_{\eta^{\mathsf{F}}}\right]\!\right] &\triangleq \leftarrow_{\eta^{\mathsf{F}}} =_{\eta_\mu cc_\mu} \end{aligned}$$

Proof. First, note that co-value substitution commutes with translation as

- $NK[\![v]\!]_\delta \{NK[\![E]\!]_\delta/\langle\Box\|\alpha\rangle\} = NK[\![v\{E/\alpha\}]\!]_\delta$,
- $NK[\![c]\!]_\delta \{NK[\![E]\!]_\delta/\langle\Box\|\alpha\rangle\} = NK[\![c\{E/\alpha\}]\!]_\delta$, and
- $NK[\![e]\!]_\delta[v] \{NK[\![E]\!]_\delta/\langle\Box\|\alpha\rangle\} = NK[\![e\{E/\alpha\}]\!]_{\{NK[\![E]\!]_\delta/\langle\Box\|\alpha\rangle\}}$,

which follows by induction on the syntax of commands, terms, and co-terms, where the base case is:

$$\begin{aligned}
NK[\![\alpha]\!]_\delta[v] \{NK[\![E]\!]_\delta/\langle\Box\|\alpha\rangle\} &= \langle v\|\alpha\rangle \{NK[\![E]\!]_\delta/\langle\Box\|\alpha\rangle\} \\
&= NK[\![E]\!]_\delta[v\{NK[\![E]\!]_\delta/\langle\Box\|\alpha\rangle\}] \\
&= NK[\![\alpha\{E/\alpha\}]\!]_\delta[v\{NK[\![E]\!]_\delta/\langle\Box\|\alpha\rangle\}]
\end{aligned}$$

Likewise, value substitution commutes with translation as $NK[\![v]\!]_\delta \{NK[\![V]\!]_\delta/x\} = NK[\![v\{V/x\}]\!]_\delta$, and so on.

The cases for η_μ, β^{F}, and η^{F} are calculations, and the cases for $\mu_{\mathcal{S}}$, $\tilde{\mu}_{\mathcal{S}}$, $\beta_{\mathcal{S}}$, and $\nu^{\forall_{<\mathsf{Ord}}}$ follow by the above commutation of substitution and translation similarly as in Lemma 9.5. The remaining case for $\eta_{\tilde{\mu}}$ is similar to Lemma 9.5 and follows from the fact that for every covalue E, $NK[\![E]\!]_\delta = \langle E'\|\beta\rangle$ for some co-value context E' and co-variable β according the two cases coming from the strong focalization of $\mathcal{S}$:

$$\begin{aligned}
NK[\![\tilde{\mu}x.\,\langle x\|E\rangle]\!]_\delta[v] &= \langle \mathbf{let}\,x = v\,\mathbf{in}\,\mu\delta.NK[\![E]\!]_\delta[x]\|\delta\rangle \\
&= \langle \mathbf{let}\,x = v\,\mathbf{in}\,\mu\delta.\,\langle E'[x]\|\beta\rangle\|\delta\rangle \\
&=_{cc_\mu} \langle \mu\delta.\,\langle \mathbf{let}\,x = v\,\mathbf{in}\,E'[x]\|\beta\rangle\|\delta\rangle \\
&=_{\mu_\alpha} \langle \mathbf{let}\,x = v\,\mathbf{in}\,E'[x]\|\beta\rangle \\
&=_{\eta_{letNK[\![\mathcal{S}]\!]}} \langle E'[x]\|\beta\rangle \\
&= NK[\![E]\!]_\delta[x]
\end{aligned}$$

$$\begin{aligned}
NK[\![\tilde{\mu}x.\,\langle x\|\tilde{\mu}y.c\rangle]\!]_\delta[v] &= \langle \mathbf{let}\,x = v\,\mathbf{in}\,\mu\delta.\,\langle \mathbf{let}\,y = x\,\mathbf{in}\,\mu\delta.NK[\![c]\!]_\delta\|\delta\rangle\|\delta\rangle \\
&=_{let_{NK[\![\mathcal{S}]\!]}} \langle \mathbf{let}\,x = v\,\mathbf{in}\,\mu\delta.\,\langle \mu\delta.NK[\![c]\!]_\delta\{x/y\}\|\delta\rangle\|\delta\rangle \\
&=_{\mu_\alpha} \langle \mathbf{let}\,x = v\,\mathbf{in}\,\mu\delta.NK[\![c]\!]_\delta\{x/y\}\|\delta\rangle \\
&=_\alpha \langle \mathbf{let}\,y = v\,\mathbf{in}\,\mu\delta.NK[\![c]\!]_\delta\|\delta\rangle \\
&= NK[\![\tilde{\mu}y.c]\!]_\delta[v] \qquad \Box
\end{aligned}$$

Lemma 9.11 (Natural deduction round trip)**.** *For any $\mathcal{T}$,* *a) if v is ς-normal then $NK[\![LK[\![v]\!]]\!]_\delta =_{\mu_\alpha\eta_\mu cc_\alpha} v$, and*

b) if c is ς-normal then $NK[\![LK[\![c]\!]]\!]_\delta =_{\mu_\alpha\eta_\mu cc_\alpha} c$.

Proof. By mutual induction on the syntax of commands, terms, and co-terms. The cases for variables, μ-abstractions, data structures, and cuts follow immediately by the inductive hypothesis, while the remaining cases are:

- $\mathbf{let}\,x = v'\,\mathbf{in}\,v$:

$$\begin{aligned}
NK[\![LK[\![\mathbf{let}\,x = v'\,\mathbf{in}\,v]\!]]\!]_\delta &= NK[\![\mu\alpha.\,\langle LK[\![v']\!] \| \tilde{\mu}x.\,\langle LK[\![v]\!] \| \alpha\rangle\rangle]\!]_\delta \\
&=_{IH} \mu\alpha.\,\langle \mathbf{let}\,x = v'\,\mathbf{in}\,\mu\delta.\,\langle v\|\alpha\rangle \| \delta\rangle \\
&=_{cc_\mu} \mu\alpha.\,\langle \mu\delta.\,\langle \mathbf{let}\,x = v'\,\mathbf{in}\,v \| \alpha\rangle \| \delta\rangle \\
&=_{\mu_\alpha} \mu\alpha.\,\langle \mathbf{let}\,x = v'\,\mathbf{in}\,v \| \alpha\rangle \\
&=_{\eta_\mu} \mathbf{let}\,x = v'\,\mathbf{in}\,v
\end{aligned}$$

- $\mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}$:

$$\begin{aligned}
&NK\left[\!\!\left[LK\left[\!\!\left[\mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}\right]\!\!\right]\right]\!\!\right]_\delta \\
&= NK\left[\!\!\left[\mu\alpha.\left\langle LK[\![v']\!] \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{\overrightarrow{Y:l}}(\vec{x}).\langle LK[\![v]\!] \| \alpha\rangle}\right]\right\rangle\right]\!\!\right]_\delta \\
&=_{IH} \mu\alpha.\left\langle \mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow \mu\delta.\,\langle v\|\alpha\rangle} \middle\| \delta\right\rangle \\
&=_{cc_\mu} \mu\alpha.\left\langle \mu\delta.\left\langle \mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} \middle\| \alpha\right\rangle \middle\| \delta\right\rangle \\
&=_{\mu_\alpha} \mu\alpha.\left\langle \mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v} \middle\| \alpha\right\rangle \\
&=_{\eta_\mu} \mathbf{case}\,v'\,\mathbf{of}\,\overrightarrow{\mathsf{K}(\overrightarrow{Y{:}l}, \vec{x}) \Rightarrow v}
\end{aligned}$$

- $\lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\right\}$:

$$\begin{aligned}
NK\left[\!\!\left[LK\left[\!\!\left[\lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\right\}\right]\!\!\right]\right]\!\!\right]_\delta &= NK\left[\!\!\left[\mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y:l}}[\vec{x}, \alpha].\langle LK[\![v]\!] \| \alpha\rangle}\right)\right]\!\!\right]_\delta \\
&=_{IH} \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow \mu\alpha.\,\langle v\|\alpha\rangle}\right\} \\
&=_{\eta_\mu} \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y{:}l}, \vec{x}] \Rightarrow v}\right\}
\end{aligned}$$

– $v' \ \mathsf{O}[\vec{B}, \vec{V}]$:

$$\begin{aligned} NK\left\llbracket LK\left\llbracket v' \ \mathsf{O}[\vec{B}, \vec{V}]\right\rrbracket\right\rrbracket_\delta &= NK\left\llbracket \mu\alpha. \left\langle LK\llbracket v' \rrbracket \middle\| \mathsf{O}^{\vec{B}}[\overrightarrow{LK\llbracket V\rrbracket}, \alpha]\right\rangle\right\rrbracket_\delta \\ &=_{IH} \mu\alpha. \left\langle v' \ \mathsf{O}[\vec{B}, \vec{V}] \middle\| \alpha\right\rangle \\ &=_{\eta_\mu} v' \ \mathsf{O}[\vec{B}, \vec{V}] \end{aligned}$$

□

Lemma 9.12 (Sequent calculus round trip)**.** *For any $\mathcal{S}$,*

a) if v is ς-normal then $LK\llbracket NK\llbracket v\rrbracket_\delta\rrbracket =_{\mu_\mathcal{S}} v$,

b) if c is ς-normal then $LK\llbracket NK\llbracket c\rrbracket_\delta\rrbracket =_{\mu_\mathcal{S}} c$,

c) if e is ς-normal then for all v, $LK\llbracket NK\llbracket e\rrbracket_\delta[v]\rrbracket =_{\mu_\mathcal{S}} \langle LK\llbracket v\rrbracket \| e\rangle$

Proof. By mutual induction on the syntax of commands, terms, and co-terms. The cases for cuts, (co-)variables, μ-abstractions, and data structures follow immediately by the inductive hypothesis, while the remaining cases are:

– $\mu\left(\overrightarrow{\mathsf{O}^{Y:l}[\vec{x}, \alpha].c}\right)$:

$$\begin{aligned} LK\left\llbracket NK\left\llbracket \mu\left(\overrightarrow{\mathsf{O}^{Y:l}[\vec{x}, \alpha].c}\right)\right\rrbracket_\delta\right\rrbracket &= LK\left\llbracket \lambda\left\{\overrightarrow{\mathsf{O}[\overrightarrow{Y:l}, \vec{x}] \Rightarrow \mu\alpha.NK\llbracket c\rrbracket_\delta}\right\}\right\rrbracket \\ &=_{IH} \mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y:l}}[\vec{x}, \alpha].\langle \mu\alpha.c \| \alpha\rangle}\right) \\ &=_{\mu_\alpha} \mu\left(\overrightarrow{\mathsf{O}^{\overrightarrow{Y:l}}[\vec{x}, \alpha].c}\right) \end{aligned}$$

– $\tilde{\mu}x.c$:

$$\begin{aligned} LK\llbracket NK\llbracket \tilde{\mu}x.c\rrbracket_\delta[v]\rrbracket &= LK\llbracket \langle \mathbf{let}\, x = v \,\mathbf{in}\, \mu\delta.NK\llbracket c\rrbracket_\delta \| \delta\rangle\rrbracket \\ &=_{IH} \langle \mu\alpha.\, \langle LK\llbracket v\rrbracket \| \tilde{\mu}x.\, \langle \mu\delta.c \| \alpha\rangle\rangle \| \delta\rangle \\ &=_{\mu_\alpha} \langle LK\llbracket v\rrbracket \| \tilde{\mu}x.c\rangle \end{aligned}$$

- $\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y:l}(\vec{x}).c}\right]$:

$$\begin{aligned} LK\left[\!\!\left[NK\left[\!\!\left[\tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y:l}(\vec{x}).c}\right]\right]\!\!\right]_{\delta}[v]\right]\!\!\right] &= LK\left[\!\!\left[\left\langle \textbf{case}\, v\, \textbf{of}\, \overrightarrow{\mathsf{K}(\overrightarrow{Y:l}, \vec{x}) \Rightarrow \mu\delta.NK[\![c]\!]_{\delta}} \middle\| \delta \right\rangle\right]\!\!\right] \\ &=_{IH} \left\langle \mu\alpha. \left\langle LK[\![v]\!] \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y:l}(\vec{x}).\langle\mu\delta.c\|\alpha\rangle}\right]\right\rangle \middle\| \delta \right\rangle \\ &=_{\mu_\alpha} \left\langle LK[\![v]\!] \middle\| \tilde{\mu}\left[\overrightarrow{\mathsf{K}^{Y:l}(\vec{x}).c}\right]\right\rangle \end{aligned}$$

- $\mathsf{O}^{\vec{B}}[\vec{V}, E]$:

$$\begin{aligned} LK\left[\!\!\left[NK\left[\!\!\left[\mathsf{O}^{\vec{B}}[\vec{V}, E]\right]\!\!\right]_{\delta}[v]\right]\!\!\right] &= LK\left[\!\!\left[NK[\![E]\!]_{\delta}[v\ \ \mathsf{O}[\vec{B}, \overrightarrow{NK[\![V]\!]_{\delta}}]]\right]\!\!\right] \\ &=_{IH} \left\langle \mu\alpha. \left\langle LK[\![v]\!] \middle\| \mathsf{O}^{\vec{B}}[\vec{V}, \alpha]\right\rangle \middle\| E \right\rangle \\ &=_{\mu_{\mathcal{S}}} \left\langle LK[\![v]\!] \middle\| \mathsf{O}^{\vec{B}}[\vec{V}, E]\right\rangle \end{aligned}$$

□

Theorem 9.6 (Parametric equational correspondence)**.** *For any stable substitution strategies $\mathcal{S}$ and $\mathcal{T} = NK[\![\mathcal{S}]\!]$, the $\lambda\mu let_{\mathcal{T}}$-calculus is in equational correspondence with the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus.*

Proof. Analogous to the proof of Theorem 9.3, relying on Lemmas 9.9, 9.10, 9.11 and 9.12 and the fact that ς-normalization still commutes with all other reductions. □

CHAPTER X

Conclusion

This dissertation has explored how a logical foundation based on the classical sequent calculus can be used to address issues in the design, theory, and implementation of programming languages, especially functional ones. We looked at how all the usual logical connectives that we study individually in the theory of computation are all instances of the general idea of *data* and *co-data*. We also looked at how the evaluation strategy of a programming language, which is usually inextricably woven into its very being, can be abstracted away as a parameter to the language description in terms of their impact on substitution, so that we can reason about the behavior of programs by just considering what variables and co-variables might stand for. This treatment of evaluation strategy extends beyond simplistic call-by-value and call-by-name evaluation to more complex evaluation methods like call-by-need, and also provides a general mechanism for mixing two or more different evaluation strategies within a single program.

Next, we looked at how the simple framework of data and co-data could be extended to more advanced typed features found in programming languages and proof assistants. In particular, type abstraction and polymorphism (in the form of either generics or modules) is represented as hidden types contained within (co-)data structures. Additionally, well-founded recursive types (like lists, trees, and streams) can be represented by (co-)data indexed with a measure of their size, whose definition follows by a well-founded recursion scheme over that index. Well-founded recursive programs over those types are then represented by quantifying over the size of the type. This technique allows for multiple different recursion schemes with their own advantages and disadvantages. Here, we considered both *noetherian* recursion which prevents programs from depending on the size index so that they can be erased at run-time, as well as *primitive* recursion which allows programs to depend on and react to the size index at run-time, similar to GADTs or index-dependent types.

With the collection of language features in place, we then studied their theory and application to programming. On the theory side, we developed a model for the language based on the idea of orthogonality which was general enough to capture several language-wide safety features, including type safety, strong normalization, and soundness of the typed extensional equational theory used to freely reason about program behavior with respect to the untyped operational semantics used to run programs. Unlike the common formulation of such models, the one presented here was parametric in two unusual dimensions (1) the connectives used to build types in the language, and (2) the evaluation strategy used to run programs in the language. These extra dimensions of generality are due to the parametric formulation of the underlying language, and allows us to establish results for many instances of the parametric language in one fell swoop.

On the application side, we looked at some ways that the ideas developed here could apply to the compilation of programs. We used the idea of polarity to come up with a small basis of polarized types which can faithfully encode all the simple (co-)data types that could possibly be declared based on a theory of type isomorphisms. The "faithfulness" criteria that we demand of encodings is important if they are to be useful for the purpose of optimizing compilation. In particular, we should expect that exactly the same equations between programs should hold both before and after encoding, so that we do not break the semantics of programs or lose out on optimization opportunities due to leaky abstractions. To achieve these faithful encodings, we employ the general wisdom from polarized logic and the call-by-push-value paradigm that types from call-by-name and call-by-values can be represented by sprinkling polarity shifts in the appropriate places. However, because in our case we are interested in evaluation strategies beyond just these two, we introduce a family of four distinct shifts that are used for encoding the correct evaluation order into types.

Completing the circle, we come back to the natural deduction world by introducing a λ-calculus based language corresponding to everything we have done in the setting of the classical sequent calculus. The two different viewpoints are then related in two ways. First, we restrict the sequent calculus to just intuitionistic logic by limiting all sequents to a single consequence. This lets us establish a correspondence between the static and dynamic semantics of pure λ-calculus and the intuitionistic language of the sequent calculus. Second, we generalize the natural deduction language to classical logic by introducing multiple consequences to the λ-calculus, which lets us extend the correspondence by relating the full language of the classical sequent calculus with the λ-calculus with first-class control. The two-way correspondence is applicable to functional programming by showing how functional programs based on the λ-calculus can be compiled to the sequent calculus, and also revealing how ideas born in the sequent calculus can be translated back into their analog in functional programming. The classical correspondence also has the additional application in optimizing compilation by giving a more expressive representation of control that can maintain sharing under optimization, avoiding the need to duplicate code while transforming programs to reveal simplifications.

Future Work

Types

We considered some advanced type features found in programming languages and proof assistants that go beyond just simple types. However, there are still many more advanced type features that could be included in the analysis and framework presented here.

Sub-typing

Intuitively, co-data types are a lot like objects and interfaces from object-oriented languages removed from assumptions about mutable state. But we would be remiss to not mention *sub-typing* which is an essential feature in any statically-typed object-oriented language. For example, if we declare an interface, we should be able to *extend* that interface with additional messages, so that any object following the extended interface can just as well be considered an object of the original one.

In the language of co-data, this idea corresponds to extending a co-data declaration with additional alternatives. For example, given our usual simple definition of the binary product co-data type $A \mathbin{\&} B$

$$\begin{aligned}&\mathbf{codata}\, X \mathbin{\&} Y \,\mathbf{where}\\ &\quad \pi_1 : \ \mid X \mathbin{\&} Y \vdash X\\ &\quad \pi_2 : \ \mid X \mathbin{\&} Y \vdash Y\end{aligned}$$

we can extend binary products to ternary products by extending the co-data declaration of & to a co-data declaration that includes an additional projection observation as follows:

$$\begin{aligned}&\textbf{codata}\,\mathsf{Tern}(X,Y,Z)\,\textbf{extends}\,X\,\&\,Y\,\textbf{where}\\&\quad\pi_3 : \ \mid \mathsf{Tern}(X,Y,Z) \vdash Z\end{aligned}$$

So the type $\mathsf{Tern}(A_1, A_2, A_3)$ includes the same projections $\pi_1\,[e_1]$ and $\pi_2\,[e_2]$ (where $e_i : A_i$) as $A_1 \oplus A_2$ does, but also includes the third projection $\pi_3\,[e_3]$. That means an object of $\mathsf{Tern}(A_1, A_2, A_3)$ must respond to all three messages, as in the term $\mu(\pi_1\,[\alpha_1].c_1 \mid \pi_2\,[\alpha_2].c_2 \mid \pi_3\,[\alpha_3].c_3)$, which is an extension of the objects of type $A \,\&\, B$. The dual of this concept of sub-typing for co-data is sub-typing for data. For example, the simple sum data type that we have used

$$\begin{aligned}&\textbf{data}\,X \oplus Y\,\textbf{where}\\&\quad\iota_1 : X \vdash X \oplus Y \mid\\&\quad\iota_2 : Y \vdash X \oplus Y \mid\end{aligned}$$

can be extended with an additional third alternative as follows:

$$\begin{aligned}&\textbf{data}\,\mathsf{OneOfThree}(X,Y,Z)\,\textbf{extends}\,X \oplus Y\,\textbf{where}\\&\quad\iota_3 : Z \vdash \mathsf{OneOfThree(X,Y,Z)} \mid\end{aligned}$$

So the type $\mathsf{OneOfThree}(A_1, A_2, A_3)$ includes the same injections $\iota_1\,(v_1)$ and $\iota_2\,(v_2)$ (where $v_i : A_i$) as $A_1 \oplus A_2$ does, but also includes the third injection $\iota_3\,(v_3)$. That means that case analysis of $\mathsf{oneOfThree}(A_1, A_2, A_3)$ must consider all three possible cases, as in the co-term $\tilde{\mu}[\iota_1\,(x_1).c_1 \mid \iota_2\,(x_2).c_2 \mid \iota_3\,(x_3).c_3]$.

The main purpose of sub-typing is not extension but *subsumption*. That is, we expect to be able to use values of a sub-type in contexts expecting the super-type. Using the above example, we would have the sub-typing relationships (written as $A <: B$ and read as "A is a subtype of B") $\mathsf{Tern}(A, B, C) <: A \,\&\, B$ and $A \oplus B <: \mathsf{OneOfThree}(A, B, C)$, which can then be used to coerce between terms and co-terms of sub-types. This generalizes the notion of type equality from Chapter VI to a directed (i.e. non-symmetric) relationship, likewise generalizing the type conversion rules to the following pair of *subsumption* rules:

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta \quad \Theta \vdash_{\mathcal{G}} A :< B : k}{\Gamma \vdash^{\Theta}_{\mathcal{G}} v : B \mid \Delta}\,SubR \qquad \frac{\Theta \vdash_{\mathcal{G}} A <: B : k \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} e : B \mid \Delta}{\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta}\,SubR$$

The right subsumption rule is what we're used to seeing in programming languages. It says that if a term v produces a result of type A which is a sub-type of B, then v can also be said to produce a result of type B. The left subsumption rule is a little different, though, and runs in the opposite direction due to duality. It says that if a co-term e consumes any result of type B which is a super-type of A, then e can also be said to consume any result of type A. These rules give the usual notion of sub-type polymorphism as coercion. We could also allow for abstraction over sub-types, analogous to the system $\mathrm{F}_{<:}$ Cardelli *et al.* (1994) extension of system F with sub-typed bounded polymorphism, by adding the kind $<_k A$ of all subtypes of A in kind k like we did for orderings in Ord.

Also note that the orthogonality models for programs in Chapter VII are already set up to easily accomodate sub-typing. In particular, recall that we used the containment ordering relation, $\mathbb{A} \sqsubseteq \mathbb{B}$ on interactions spaces which means that everything in $\mathbb{A}$ is contained in $\mathbb{B}$. However, there is another natural ordering relationship between interaction spaces which flips directions between the two sides of the space. That is, given any spaces $\mathbb{A} = (\mathbb{A}_+, \mathbb{A}_-)$ and $\mathbb{B} = (\mathbb{B}_+, \mathbb{B}_-)$ (so that $\mathbb{A}_+$ and $\mathbb{B}_+$ are the positive sides of $\mathbb{A}$ and $\mathbb{B}$ and $\mathbb{A}_-$ and $\mathbb{B}_-$ are the negative sides), we have the *sub-space* relation $\mathbb{A} \leq \mathbb{B}$ defined as

$$(\mathbb{A}_+, \mathbb{A}_-) \leq (\mathbb{B}_+, \mathbb{B}_-) \triangleq (\mathbb{A}_+ \subseteq \mathbb{B}_+) \wedge (\mathbb{B}_+ \subseteq \mathbb{A}_-)$$

This follows the intuitive meaning of behavioral sub-typing (Liskov, 1987), where A is a sub-type of B if every value (i.e. positive element) of A can be used in any context (i.e. negative element) expecting a B, or conversely every context of B can be used with any value of A.

Implicit types

The treatment of connectives that we have explored in this dissertation has been very syntax directed. That is to say, the typing rules specific to every connective only applies to expressions of certain syntactic forms. For example, the right rules specific to $A \oplus B$ is to construct an injection $\iota_i\,(v) : A \otimes B$ for $i = 1$ or $i = 2$ and the only left rule specific to $A + B$ is to deconstruct an injection $\tilde{\mu}[\iota_1\,(x).c_1 \mid \iota_2\,(x).c_2] : A \oplus B$.

However, there are also more implicit forms types which are not tied to specific syntax. For example, *union* and *intersection* types combine existing types by directly combining the programs they specify without modification. More specifically, the union of two types, written $A \vee B$, is like an untagged union in C, so $v : A \vee B$ if either $v : A$ or $v : B$ without tagging it with an injection explaining which was the case. Dually, the intersection of two types, written $A \wedge B$, requires membership in both types so that $v : A \wedge B$ if both $v : A$ and $v : B$ as-is. However, note that these implicit rules are dangerous: in a language of effects, implicit union and intersection types require careful *value* and *evaluation context* restrictions, depending on the evaluation strategy of the language, similar to the value restriction in ML to be sound. In particular, the ML style of implicit polymorphism can be seen as an infinite intersection quantifier, $\bigwedge X.A$, whereas the dual form of implicit existential quantification can be seen as an infinite union, $\bigvee X.A$, which require the same restrictions as binary intersection and union types.

Understanding these value and evaluation context restrictions for intersection and union types was one of the original motivations for studying polarity in computation (Zeilberger, 2008a). And as noted by Munch-Maccagnoni (2009), the orthogonal model of programs puts these kinds of implicit types on firmer ground. In particular, these sorts of types are closely related to the orthogonal model of sub-typing mentioned previously. In Chapter VII, we used union and intersection operations on interaction spaces, $\mathbb{A} \sqcup \mathbb{B}$ and $\mathbb{A} \sqcup \mathbb{B}$ respectively, that fit well with the containment relationship. However, there is an alternative presentation of union and intersection that fits with sub-spaces. Given any spaces $\mathbb{A} = (\mathbb{A}_+, \mathbb{A}_-)$ and $\mathbb{B} = (\mathbb{B}_+, \mathbb{B}_-)$, their sub-spacial union and intersections, $\mathbb{A} \vee \mathbb{B}$ and $\mathbb{A} \wedge \mathbb{B}$ respectively, are defined as

$$(\mathbb{A}_+, \mathbb{A}_-) \vee (\mathbb{B}_+, \mathbb{B}_-) \triangleq (\mathbb{A}_+ \cup \mathbb{B}_+, \mathbb{A}_- \cap \mathbb{B}_-)$$
$$(\mathbb{A}_+, \mathbb{A}_-) \wedge (\mathbb{B}_+, \mathbb{B}_-) \triangleq (\mathbb{A}_+ \cap \mathbb{B}_+, \mathbb{A}_- \cup \mathbb{B}_-)$$

Note that in combination with the containment-based operations, these complete the set, giving all four combinations of union and intersection of the two sides of interaction spaces. And like the containment operations (Property 7.3), these operations share similar de Morgan laws with respect to sub-spaces. This lets us build a model for binary or infinite intersection and union types similar to Munch-Maccagnoni (2009) as the appropriately value-restricted positively or negatively constructed types from Chapter VII.

The practical application of these more implicit types is that they allow for more sub-typing and type equalities since they leave no trace on the underlying programs, and thus enable more programs to be written. For example, there is Girard's "shocking" Munch-Maccagnoni (2009) type equality $\bigwedge X.(A \oplus B) = (\bigwedge X.A) \oplus (\bigwedge X.B)$ which does not make sense if we interpret $\bigwedge$ as a traditional universal quantification—just because either A or B is true for any X doesn't mean that it must always be that A is true for every X or B is true for every X—but make sense when viewed as a cost-free run-time type coercion. So these kinds of implicit types could be useful in applications that demand both very expressive types with little to no run-time cost.

Dependent types

One of the most active research topics in type theory for programming languages is *dependent types*: allowing the compile-time type of a program depend on the run-time value of the program. This dissertation only considers a limited view of type dependency in the form of the Ix-indexed types from Chapter VI. The Ix indexes in programs influence their behavior, so they are a form of run-time value, but also types can depend on their value as well in the primitive recursive (co-)data declarations. That gives us a limited form of dependent types over just the natural numbers as type indexes. However, generalizing the classical sequent calculus to full-spectrum dependent types that allow for type dependency on run-time values of arbitrary other types is much more challenging.

For example, in terms of the λlet natural deduction language, consider the following dependent version of the *Let* rule:

$$\frac{\Gamma \vdash v : A \quad \Gamma, x : A \vdash v' : B}{\Gamma \vdash \mathbf{let}\, x = v\, \mathbf{in}\, v' : B\,\{v/x\}}\; Let_{dep}$$

The dependency comes from the fact that in the premise, the variable x can be seen in both the term v' as well as in the consequence B. So in the conclusion of the inference rule, we bind v to x in the term v' and also substitute v for x in the consequence $B\,\{v/x\}$.

Unfortunately, this rule does not fit well with the commuting conversions of the λlet-calculus. For example, in the call-by-value $\mathcal{V}$ instance, we have the following $cc_{\mathcal{V}}$

rule for re-associating nested **let** expressions:

$$\mathbf{let}\, y = (\mathbf{let}\, x = v \,\mathbf{in}\, v')\, \mathbf{in}\, v'' \quad \rightarrow_{cc_\nu} \mathbf{let}\, x = v \,\mathbf{in}\, (\mathbf{let}\, y = v' \,\mathbf{in}\, v'')$$

This kind of reduction is essential theoretically for the correspondence between the intuitionistic sequent calculus and natural deduction, and also has practical applications in optimizing compilers for functional programming languages. However, this reduction step is highly dubious when combined with the dependent Let_{dep} typing rule, since it changes the types of terms. Before reduction, suppose that we have the following typing derivation using Let_{dep}:

$$\dfrac{\dfrac{\Gamma \vdash v : A \quad \Gamma, x : A \vdash v' : B}{\Gamma \vdash \mathbf{let}\, x = v \,\mathbf{in}\, v' : B\,\{v/x\}}\,Let_{dep} \qquad \Gamma, y : B\,\{v/x\} \vdash v'' : C}{\Gamma \vdash \mathbf{let}\, y = (\mathbf{let}\, x = v \,\mathbf{in}\, v')\, \mathbf{in}\, v'' : C\,\{(\mathbf{let}\, x = v \,\mathbf{in}\, v')/y\}}\,Let_{dep}$$

Yet, after reduction, the re-associated typing derivation sees different dependencies on values in types:

$$\dfrac{\Gamma \vdash v : A \qquad \dfrac{\Gamma, x : A \vdash v' : B \quad \Gamma, x : A, y : B \vdash v'' : C}{\Gamma, x : A \vdash \mathbf{let}\, y = v' \,\mathbf{in}\, v'' : C\,\{v'/y\}}\,Let_{dep}}{\Gamma \vdash \mathbf{let}\, x = v \,\mathbf{in}\, \mathbf{let}\, y = v' \,\mathbf{in}\, v'' : C\,\{v'/y\}}\,Let_{dep}$$

In contrast, in the sequent calculus we do not have this form of elimination rule, which pairs an object of interest with some observation on it, but rather input-independent left rules. The **let** expression above corresponds to an input abstraction typed by the left rule

$$\dfrac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\,AL$$

which is then later paired with its input via a *Cut* as in

$$\dfrac{\Gamma' \vdash v : A \mid \Delta' \qquad \dfrac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\,AL}{\langle v \| \tilde{\mu}x.c \rangle : (\Gamma, \Gamma' \vdash \Delta, \Delta')}\,Cut$$

The "return type" of the binding, which was explicitly known in the **let** expression in natural deduction, instead implicitly found in the output environment Δ of the co-term $\tilde{\mu}x.c$. Forming a full-spectrum dependently typed sequent calculus would therefore involve a dependent *Cut* rule that ties together the (co-)values of the two sides of a cut to be referenced in the environments of the other side. Finding a solution to this problem would lead to a dependent type theory that is more robust to extensions, being able to accomodate effect like the control implicit in the classical sequent calculus, as well as handling additional simplifications like the commuting conversions above without altering types in troublesome ways.

Exotic types

Our exploration of the sequent calculus naturally brought up types like negation ($\sim A$ and $\neg A$) and the disjunctive co-data type ($A \parr B$), and the subtraction data type ($A - B$) which are normally unheard of in programming. Unsurprisingly, these were also exactly the sorts of types that were disregarded in the correspondence between the (classical and intuitionistic) sequent calculus and natural deduction in Chapter IX. Each of these types requires having more than one consequence to a constructer or observer, which does not have a nice, natural analogue in the language of the λ-calculus.

It would be interesting to find out if all of these types can be represented naturally in the λ-calculus, and to see if any are useful for ordinary programming or are analogous to other programming features. For example, subtraction types have been used to model delimited control (Ariola *et al.*, 2009b). Are par ($\parr$) types also related to delimited control? Do the involutive negation types give a practical model of call stacks (Munch-Maccagnoni, 2014) in the implementation of functional programming?

Effects

Languages based on the classical sequent calculus include an inherent notion of first-class control effect without adding anything extra. However, there are many more kinds of effects that arise in programming languages which could be analysed from the sequent calculus point of view.

General recursion

In this dissertation, we considered recursion as a language feature, but only the well-founded kind which could not lead to infinite loops. But general purpose programming languages include general recursion, which accepts the possibility for infinite loops in exchange for greater expressive power. General recursion could be added to the $\mu\tilde{\mu}$ sequent calculus by including two new types of general binders, $\nu x.V$ and $\tilde{\nu}\alpha.E$, which give a self-referential name to values and co-values. These self-reference abstractions can be unrolled at run-time as follows:

$$(\nu_{\mathcal{S}}) \qquad \nu x.V \succ_{\nu_{\mathcal{S}}} V\{(\nu x.V)/x\} \qquad\qquad (\tilde{\nu}_{\mathcal{S}}) \qquad \tilde{\nu}\alpha.E \succ_{\tilde{\nu}_{\mathcal{S}}} E\{[\tilde{\nu}\alpha.E]/\alpha\}$$

where we restrict the abstractions to (co-)values so that they are valid for substitute according to the chosen strategy. Because of this value restriction imposed at run-time, we get the following two dual typing rules for general recursion:

$$\frac{\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} V : A \mid \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \nu x.V : A \mid \Delta}\ \mathit{RecR} \qquad\qquad \frac{\Gamma \mid E : A \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta}{\Gamma \mid \tilde{\nu}\alpha.E : A \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathit{RecL}$$

Note that the general form of recursion can be seen as subsuming the well-founded recursion considered here by translating the recursive size abstractions to a general self-reference abstraction over the non-recursive pattern match as follows for the $\forall j < N.A$ and $\exists j < N.A$ types:

$$\mu(j{<}N \mathbin{@_x} \alpha.c) = \nu x.\mu(j{<}N \mathbin{@} \alpha.c)$$
$$\tilde{\mu}[j{<}N \mathbin{@_\alpha} x.c] = \tilde{\nu}\alpha.\tilde{\mu}[j{<}N \mathbin{@} x.c]$$

The above encoding of well-founded recursion as general recursion is sensible in any focalizing strategy because then the case abstractions are always (co-)values. This is another example where the correct use of data or co-data gives the appropriate evaluation order independently of the chosen evaluation strategy of the language at large.

The syntactic theory of general recursion in the sequent calculus is fairly straightforward since we must already be care of computations which don't return to their caller because of control effects, so the developments in Chapters V, VI, VIII, and IX are largely unaffected by their inclusion. The main point of interest is the semantic analysis in Chapter VII. As-is, the recursive typing rules *RecR* and *RecL* are not sound with respect to the model in Chapter VII. To be expected, general recursion imposes extra demands on such models, such as some form of continuity property of the safety condition in question which lets us justify that the recursive fixed points are safe because all their approximations (i.e. finite unrollings) are safe.

Both Pitts (2000) and Mellies & Vouillon (2005) give orthogonality-based models that incorporate general recursion. It would be interesting to isolate the minimal extra assumption that is needed to include general recursive fixed points to the parametric orthogonality models considered here.

Delimited control

The $\mu\tilde{\mu}$ sequent calculus inherits a form of classical control effects corresponding to from classical logic (Griffin, 1990), however, there is a more expressive family of control effects known as *delimited control* (Felleisen, 1988). Whereas classical control allows for the first-class manipulation of evaluation contexts, delimited control adds the ability to delimit the scope of that control, effectively breaking down the evaluation context into independent segments that can be handled separately. It turns out this extra functionality radically increases the expressive power of control, to the point that any monadic effect can be encoded directly in a call-by-value functional language using delimited control (Filinski, 1994, 1999), giving a form of user-defined effects.

We might then ask, what does delimited control look like in the sequent calculus? One way to extend the sequent calculus with delimited control effects is to add a *dynamic*, as opposed to static, co-variable $\widehat{\mathsf{tp}}$ (Munch-Maccagnoni, 2014), short for the "top-level", in analogy to the type-theoretic account of delimited control in the λ-calculus (Ariola *et al.*, 2009b). The main idea of the dynamic co-variable acts as a delimiter by making the η law for the $\widehat{\mathsf{tp}}$ abstraction

$$\mu\widehat{\mathsf{tp}}.\left\langle V \middle\| \widehat{\mathsf{tp}} \right\rangle \to V$$

fire *even if* $\widehat{\mathsf{tp}}$ is "free" in V, but only when V is a value. Another approach would be to collapse the distinction between terms and commands based on the syntactic relaxation of the $\lambda\mu$-calculus (Herbelin & Ghilezan, 2008; Downen & Ariola, 2014a). By collapsing terms and commands, we can effectively "stack" many pending co-values in a single command with a chain of cuts as in

$$\langle\langle\langle\langle v\|\alpha\rangle\|\beta\rangle\|\gamma\rangle\|\delta\rangle$$

so that each cut acts as a delimiter separating the co-values α, β, γ, and δ which are seen by the active term v in that order.

Both of these approaches have some unfortunate weaknesses in their current state, however. First, both views of delimited control present an rather asymmetric language construct, which goes against the symmetry of the sequent calculus seen in every other language feature. Why is there only a dynamic co-variable, and no dynamic variable with the dual behavior? And why are only terms collapsed with commands, instead of both terms and co-terms, so that the pending stack of cuts can grow in both directions simultaneously? Both of these extensions to dualize delimited control would bring it more into the dual character of the sequent calculus, and avoid our natural bias from the λ-calculus to only think of terms first while co-terms are an afterthought at best. Second, the classical $\mu\tilde{\mu}$-calculus has a canonical type system that corresponds to the LK system of classical logic. However, the type system for $\widehat{\mathsf{tp}}$ in the sequent calculus given by Munch-Maccagnoni (2014) is, while also canonical, too restrictive to make any interesting use of extra capabilities of the delimited control effect. And the approach of collapsing commands and terms by Herbelin & Ghilezan (2008); Downen & Ariola (2014a) is effectively untyped. While there are expressive type systems for delimited control, they are behavioral, meaning they depend on the specific evaluation order of the language, and are type-and-effect systems that are designed specifically for those programming languages rather than coming from logic like LK or the λ- and $\lambda\mu$-calculi. This poses the question: what is the logical concept that corresponds with delimited control, and how does it integrate with duality?

Algebraic effect handlers

Another technique for adding user-defined effects to a language is with *algebraic effect handlers* (Cartwright & Felleisen, 1994; Pretnar, 2010; Bauer & Pretnar, 2015). Effect handlers act as one or more "administrators" that handle requests for primitive effectful operations and dictate how to evaluate them at run-time. For example, we could describe mutable state (with a single reference cell) by the *get* and *put* operations. A program could then request to *get* the current state or *put* a new one, and the handler of that program is responsible for interpreting those requests and putting them to action on the stateful reference cell.

Data and co-data in the sequent calculus may be seen as a way of describing the interface of interaction between the program and the administrator. In particular, the dual nature of data and co-data suggests two dual ways of describing the third party that defines an effect: *third-party as an administrator*, where there is an external, abstract definition for the behavior of primitive operations and the program must yield control of execution when reaching an effectful operation, or *third-party as a resource*, where the program carries around extra, external information that may be scrutinized in order to perform arbitrary effect-dependent behavior. The sequent calculus already gives a rich language for interacting with contexts described as structures and abstract processes; extending the sequent calculus with a notion of algebraic effects would extend this language to programmable computational effects, similar to Felleisen's proposal for abstract continuations (Felleisen *et al.*, 1988).

The fact that both these interpretations of algebraic effect handlers rely on some third-party is reminiscent of the approach to delimited control of directly chaining many *Cut*s by collapsing terms or co-term with commands. The essence appears to be that, while only one party is needed to represent general recursion and two parties are needed to represent classical control, other effects like delimited control or effect handlers need (at least) three parties to carry out effectful computation. This also suggests an approach to generalizing the orthogonality models from Chapter VII to give a unified framework for admitting other kinds of effects. If we can compose more than two entities into the elements of a computational pole, then we could model the inclusion of heaps and handlers in the state of computation needed for representing effects like mutable state and exceptions.

Linearity

While this dissertation has primarily been about languages based on classical logic, there is still an inherent implicit connection to linear logic (Girard, 1987). In particular, the dual roles of construction and deconstruction via patterns and pattern matching gives yet another view of the logical rules of linear logic. Indeed, we used this analogy to name the two different data and co-data forms of conjunction ($\otimes$ and $\&$) and disjunction ($\oplus$ and $\parr$) based on this connection with the multiplicative and additive connectives of linear logic. However, since we assume that the structural rules of weakening and contraction are always valid, we end up with a system equivalent to classical logic in terms of provability (the ability to build a derivation for a given judgement). To build a stronger connection with linear logic, we would need a more refined type system which lets us rule out certain applications of structural rules.

From the standpoint of programming languages, incorporating linearity in the type system would give a more general account of purity than was given in Chapter IX.

That is, linearity as a programming feature is about restricting effects, rather than enabling them. In particular, we established a pure subset of the classical sequent calculus (based on Gentzen's LK) by restricting everything to exactly one consequence (as in Gentzen's LJ). The effect of this restriction is to ensure that every co-value is used exactly once during execution (barring other effects like infinite loops). In contrast, linear logic allows for multiple consequences while still maintaining this same run-time invariant, as noted by Girard (1987). Thus, we can still enforce properties like referential transparency while also accommodating more exotic types like $A \parr B$ which step outside the normal functional paradigm. A language based on Girard's (1993) logic of unity—which integrates and subsumes each of classical, intuitionistic, and linear logics—would let us safely integrate first-class control with pure functional programming, combining the advantages of both.

REFERENCES CITED

Abadi, Martín, & Cardelli, Luca. (1996). *A theory of objects*. 1st edn. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Abel, Andreas. (2006). *A polymorphic lambda calculus with sized higher-order types*. Ph.D. thesis, Ludwig-Maximilians-Universität München.

Abel, Andreas, Pientka, Brigitte, Thibodeau, David, & Setzer, Anton. (2013). Copatterns: Programming infinite structures by observations. *Pages 27–38 of: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. POPL '13. New York, NY, USA: ACM.

Abel, Andreas M., & Pientka, Brigitte. (2013). Wellfounded recursion with copatterns: A unified approach to termination and productivity. *Pages 185–196 of: Proceedings of the 18th ACM SIGPLAN international conference on functional programming*. ICFP '13. New York, NY, USA: ACM.

Andreoli, Jean-Marc. (1992). Logic programming with focusing proofs in linear logic. *Journal of logic and computation*, **2**(3), 297–347.

Appel, Andrew W. (1992). *Compiling with continuations*. New York, NY, USA: Cambridge University Press.

Ariola, Zena M., & Felleisen, Matthias. (1997). The call-by-need lambda calculus. *Journal of functional programming*, **7**(3), 265–301.

Ariola, Zena M., & Herbelin, Hugo. (2003). Minimal classical logic and control operators. *Pages 871–885 of:* Baeten, Jos C. M., Lenstra, Jan Karel, Parrow, Joachim, & Woeginger, Gerhard J. (eds), *Automata, languages and programming: 30th international colloquium*. ICALP 2003. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ariola, Zena M., & Herbelin, Hugo. (2008). Control reduction theories: The benefit of structural substitution. *Journal of functional programming*, **18**(3), 373–419.

Ariola, Zena M., Maraist, John, Odersky, Martin, Felleisen, Matthias, & Wadler, Philip. (1995). A call-by-need lambda calculus. *Pages 233–246 of: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages*. POPL '95. New York, NY, USA: ACM.

Ariola, Zena M., Bohannon, Aaron, & Sabry, Amr. (2009a). Sequent calculi and abstract machines. *ACM transactions on programming languages and systems*, **31**(4), 13:1–13:48.

Ariola, Zena M., Herbelin, Hugo, & Sabry, Amr. (2009b). A type-theoretic foundation of delimited continuations. *Higher-order and symbolic computation*, **22**(3), 233–273.

Ariola, Zena M., Herbelin, Hugo, & Saurin, Alexis. (2011). Classical call-by-need and duality. *Pages 27–44 of: Typed lambda calculi and applications: 10th international conference*. TLCA'11. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ariola, Zena M., Downen, Paul, Herbelin, Hugo, Nakata, Keiko, & Saurin, Alexis. (2012). Classical call-by-need sequent calculi: The unity of semantic artifacts. *Pages 32–46 of:* Schrijvers, Tom, & Thiemann, Peter (eds), *Functional and logic programming: 11th international symposium*. Lecture Notes in Computer Science, vol. 7294. Berlin, Heidelberg: Springer Berlin Heidelberg.

Barbanera, Franco, & Berardi, Stefano. (1994). A symmetric lambda calculus for "classical" program extraction. *Pages 495–515 of: Proceedings of the international conference on theoretical aspects of computer software*. TACS '94. London, UK, UK: Springer-Verlag.

Barendregt, Hendrik Pieter. (1985). *The lambda calculus: Its syntax and semantics*. Studies in Logic and the Foundations of Mathematics. Amsterdam, New-York, Oxford: North-Holland.

Bauer, Andrej, & Pretnar, Matija. (2015). Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming*, **84**(1), 108–123.

Cardelli, Luca, Martini, Simone, Mitchell, John C., & Scedrov, Andre. (1994). An extension of system F with subtyping. *Information and computation*, **109**(1), 4–56.

Carraro, Alberto, Ehrhard, Thomas, & Salibra, Antonino. (2012). The stack calculus. *Pages 93–108 of: Proceedings seventh workshop on logical and semantic frameworks, with applications*. LSFA 2012.

Cartwright, Robert, & Felleisen, Matthias. (1994). Extensible denotational language specifications. *Pages 244–272 of:* Hagiya, Masami, & Mitchell, John C. (eds), *Theoretical aspects of computer software: International symposium*. TACS '94. Berlin, Heidelberg: Springer Berlin Heidelberg.

Church, Alonzo. (1932). A set of postulates for the foundation of logic. *Annals of mathematics*, **33**(2), 346–366.

Church, Alonzo. (1936). An unsolvable problem of elementary number theory. *American journal of mathematics*, **58**(2), 345–363.

Coq 8.4. (2012). *The Coq proof assistant reference manual.* version 8.4 edn. INRIA.

Coquand, Thierry. (1985). *Une théorie des constructions.* Ph.D. thesis, Université Paris 7.

Cousineau, Guy, Curien, Pierre-Louis, & Mauny, Michel. (1987). The categorical abstract machine. *Science of computer programming*, **8**(2), 173–202.

Curien, Pierre-Louis, & Herbelin, Hugo. (2000). The duality of computation. *Pages 233–243 of: Proceedings of the fifth ACM SIGPLAN international conference on functional programming.* ICFP '00. New York, NY, USA: ACM.

Curien, Pierre-Louis, & Munch-Maccagnoni, Guillaume. (2010). The duality of computation under focus. *Pages 165–181 of:* Calude, Cristian S., & Sassone, Vladimiro (eds), *Theoretical computer science: 6th IFIP TC 1/WG 2.2 international conference, TCS 2010, held as part of WCC 2010.* TCS 2010. Berlin, Heidelberg: Springer Berlin Heidelberg.

Curry, Haskell B., Feys, Robert, & Craig, William. (1958). *Combinatory logic.* Vol. 1. North-Holland Publishing Company.

Cytron, Ron, Ferrante, Jeanne, Rosen, Barry K., Wegman, Mark N., & Zadeck, F. Kenneth. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM transactions on programming languages and systems*, **13**(4), 451–490.

Danos, Vincent, Joinet, Jean-Baptiste, & Schellinx, Harold. (1997). A new deconstructive logic: Linear logic. *Journal of symbolic logic*, **62**(3), 755—-807.

David, Rene, & Py, Walter. (2001). $\lambda\mu$-calculus and Bohm's theorem. *Journal of symbolic logic*, **66**(1), 407–413.

Davis, Martin. (2004). *The undecidable: Basic papers on undecidable propositions, unsolvable problems, and computable functions.* Dover Publication, Incorporated.

de Bruijn, Nicolaas. (1968). *AUTOMATH, a language for mathematics.* Tech. rept. 66-WSK-05. Technological University Eindhoven.

Di Cosmo, Roberto. (1995). *Isomorphisms of types: From λ-calculus to information retrieval and language design.* Basel, Switzerland: Birkhauser Verlag.

Downen, Paul, & Ariola, Zena M. (2012). A systematic approach to delimited control with multiple prompts. *Pages 234–253 of:* Seidl, Helmut (ed), *Programming languages and systems: 21st European symposium on programming, ESOP 2012, held as part of the European joint conferences on theory and practice of software, ETAPS 2012.* Lecture Notes in Computer Science, vol. 7211. Springer Berlin Heidelberg.

Downen, Paul, & Ariola, Zena M. (2014a). Compositional semantics for composable continuations: From abortive to delimited control. *Pages 109–122 of: Proceedings of the 19th ACM SIGPLAN international conference on functional programming*. ICFP '14. New York, NY, USA: ACM.

Downen, Paul, & Ariola, Zena M. (2014b). Delimited control and computational effects. *Journal of functional programming*, **24**, 1–55.

Downen, Paul, & Ariola, Zena M. (2014c). The duality of construction. *Pages 249–269 of:* Shao, Zhong (ed), *Programming languages and systems: 23rd European symposium on programming, ESOP 2014, held as part of the European joint conferences on theory and practice of software, ETAPS 2014*. Lecture Notes in Computer Science, vol. 8410. Springer Berlin Heidelberg.

Downen, Paul, Maurer, Luke, Ariola, Zena M., & Varacca, Daniele. (2014). Continuations, processes, and sharing. *Pages 69–80 of: Proceedings of the 16th international symposium on principles and practice of declarative programming*. PPDP '14. New York, NY, USA: ACM.

Downen, Paul, Johnson-Freyd, Philip, & Ariola, Zena M. (2015). Structures for structural recursion. *Pages 127–139 of: Proceedings of the 20th ACM SIGPLAN international conference on functional programming*. ICFP '15. New York, NY, USA: ACM.

Downen, Paul, Maurer, Luke, Ariola, Zena M., & Peyton Jones, Simon. (2016). Sequent calculus as a compiler intermediate language. *Pages 74–88 of: Proceedings of the 21st ACM SIGPLAN international conference on functional programming*. ICFP '16. New York, NY, USA: ACM.

Dummett, Michael. (1991). The logical basis of methaphysics. *The William James lectures, 1976*. Harvard University Press, Cambridge, Massachusetts.

Felleisen, Matthias. (1991). On the expressive power of programming languages. *Science of computer programming*, **17**(1-3), 35–75.

Felleisen, Matthias, & Friedman, Daniel P. (1986). Control operators, the SECD machine, and the λ-calculus. *Pages 193–219 of: Proceedings of the IFIP TC 2/WG2.2 working conference on formal descriptions of programming concepts part III*.

Felleisen, Matthias, & Hieb, Robert. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, **103**(2), 235–271.

Felleisen, Matthias, Wand, Mitch, Friedman, Daniel, & Duba, Bruce. (1988). Abstract continuations: A mathematical semantics for handling full jumps. *Pages 52–62 of: Proceedings of the 1988 ACM conference on LISP and functional programming.* LFP '88. New York, NY, USA: ACM.

Felleisen, Mattias. (1988). The theory and practice of first-class prompts. *Pages 180–190 of: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages.* POPL '88. New York, NY, USA: ACM.

Filinski, Andrzej. (1989). *Declarative continuations and categorical duality.* M.Phil. thesis, Computer Science Department, University of Copenhagen.

Filinski, Andrzej. (1994). Representing monads. *Pages 446–457 of: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages.* POPL '94. New York, NY, USA: ACM.

Filinski, Andrzej. (1999). Representing layered monads. *Pages 175–188 of: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages.* POPL '99. New York, NY, USA: ACM.

Fischer, Michael J. (1993). Lambda-calculus schemata. *Lisp and symbolic computation*, **6**(3-4), 259–288.

Gentzen, Gerhard. (1935a). Untersuchungen über das logische schließen. I. *Mathematische zeitschrift*, **39**(1), 176–210.

Gentzen, Gerhard. (1935b). Untersuchungen über das logische schließen. II. *Mathematische zeitschrift*, **39**(1), 405–431.

Giménez, Eduardo. (1996). *Un calcul de constructions infinies et son application a la vérification dd systèmes communicants.* Ph.D. thesis, Ecole Normale Supérieure de Lyon.

Girard, Jean-Yves. (1971). Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. *Pages 63–92 of:* Fenstad, J. E. (ed), *Proceedings of the 2nd Scandinavian logic symposium.* North-Holland.

Girard, Jean-Yves. (1987). Linear logic. *Theoretical computer science*, **50**(1), 1–101.

Girard, Jean-Yves. (1991). A new constructive logic: Classical logic. *Mathematical structures in computer science*, **1**(3), 255–296.

Girard, Jean-Yves. (1993). On the unity of logic. *Annals of pure and applied logic*, **59**(3), 201–217.

Girard, Jean-Yves. (2001). Locus solum: From the rules of logic to the logic of rules. *Mathematical structures in computer science*, **11**(3), 301–506.

Girard, Jean-Yves, Taylor, Paul, & Lafont, Yves. (1989). *Proofs and types.* New York, NY, USA: Cambridge University Press.

Gödel, Kurt. (1934). *On undecidable propositions of formal mathematical systems.* Published in Davis (2004). Lecture notes taken by Stephen C. Kleene and J. Barkley Rosser.

Gödel, Kurt. (1980). On a hitherto unexploited extension of the finitary standpoint. *Journal of philosophical logic*, **9**(2), 133–142.

Graham-Lengrand, Stéphane. (2015). *The Curry-Howard view of classical logic.* Lecture notes for the Master Parisien de Recherche en Informatique (MPRI).

Griffin, Timothy G. (1990). A formulae-as-types notion of control. *Pages 47–58 of: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on principles of programming languages.* POPL '90. New York, NY, USA: ACM.

Hagino, Tatsuya. (1987). A typed lambda calculus with categorical type constructors. *Pages 140–157 of:* Pitt, David H., Poigné, Axel, & Rydeheard, David E. (eds), *Category theory and computer science.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Hagino, Tatsuya. (1989). Codatatypes in ML. *Journal of symbolic computation*, **8**(6), 629–650.

Herbelin, Hugo. (2005). *C'est maintenant qu'on calcule : Au coeur de la dualité.* Habilitation thesis, Université Paris 11.

Herbelin, Hugo, & Ghilezan, Silvia. (2008). An approach to call-by-name delimited continuations. *Pages 383–394 of: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages.* POPL '08. New York, NY, USA: ACM.

Herbelin, Hugo, & Zimmermann, Stéphane. (2009). An operational account of call-by-value minimal and classical λ-calculus in "natural deduction" form. *Pages 142–156 of:* Curien, Pierre-Louis (ed), *Typed lambda calculi and applications: 9th international conference.* TLCA 2009. Berlin, Heidelberg: Springer Berlin Heidelberg.

Howard, William A. (1980). The formulae-as-types notion of constructions. *Pages 479–490 of: To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism.* Academic Press. Unpublished manuscript of 1969.

Huet, Gérard. (1997). The zipper. *Journal of functional programming*, **7**(5), 549–554.

Johnson-Freyd, Philip, Downen, Paul, & Ariola, Zena M. (2016). First class call stacks: Exploring head reduction. *Proceedings of the workshop on continuations, WoC 2016, London, UK, April 12th 2015.* EPTCS, vol. 212.

Johnson-Freyd, Philip, Downen, Paul, & Ariola, Zena M. (2017). Call-by-name extensionality and confluence. *Journal of functional programming*, **27**, e12.

Kay, Alan C. (1993). The early history of Smalltalk. *Pages 69–95 of: The second ACM SIGPLAN conference on history of programming languages.* HOPL-II. New York, NY, USA: ACM.

Kelsey, Richard, Clinger, William, & et al., Jonathan Rees. (1998). Revised^5 report on the algorithmic language Scheme. *Higher-order and symbolic computation*, **11**(1), 7–105.

Kennedy, Andrew. (2007). Compiling with continuations, continued. *Pages 177–190 of: Proceedings of the 12th ACM SIGPLAN international conference on functional programming.* ICFP '07. New York, NY, USA: ACM.

Klop, Jan Willem, & de Vrijer, Roel C. (1989). Unique normal forms for lambda calculus with surjective pairing. *Information and computation*, **80**(2), 97–113.

Krivine, Jean-Louis. (2007). A call-by-name lambda-calculus machine. *Higher-order and symbolic computation*, **20**(3), 199–207.

Krivine, Jean-Louis. (2009). Realizability in classical logic. *Panoramas et synthèses*, **27**, 197–229.

Lambek, Joachim, & Scott, Philip J. (1986). *Introduction to higher order categorical logic.* New York, NY, USA: Cambridge University Press.

Laurent, Olivier. 2002 (Mar.). *Étude de la polarisation en logique.* Ph.D. thesis, Université de la Méditerranée - Aix-Marseille II.

Lengrand, Stéphane, & Miquel, Alexandre. (2008). Classical F_ω, orthogonality and symmetric candidates. *Annals of pure and applied logic*, **153**(1), 3–20.

Levy, Paul Blain. (2001). *Call-by-push-value.* Ph.D. thesis, Queen Mary and Westfield College, University of London.

Levy, Paul Blain. (2003). *Call-by-push-value: A functional/imperative synthesis.* Semantics Structures in Computation, vol. 2. Springer Netherlands.

Lindley, Sam. (2005). *Normalisation by evaluation in the compilation of typed functional programming languages.* Ph.D. thesis, University of Edinburgh, College of Science and Engineering, School of Informatics.

Liskov, Barbara. (1987). Keynote address - data abstraction and hierarchy. *Pages 17–34 of: Addendum to the proceedings on object-oriented programming systems, languages and applications (addendum)*. OOPSLA '87. New York, NY, USA: ACM.

Maraist, John, Odersky, Martin, & Wadler, Philip. (1998). The call-by-need lambda calculus. *Journal of functional programming*, **8**(3), 275–317.

Martin-Löf, Per. (1975). An intuitionistic theory of types: Predicative part. *Pages 73–118 of: Logic colloquium '73*. Studies in Logic and the Foundations of Mathematics, vol. 80. Amsterdam, The Netherlands: North-Holland.

Martin-Löf, Per. (1982). Constructive mathematics and computer programming. *Pages 153–175 of: Proceedings of the sixth international congress for logic, methodology, and philosophy of science*. Amsterdam, The Netherlands: North-Holland.

Martin-Löf, Per. (1998). An intuitionistic theory of types. *Pages 127–172 of: Twenty-five years of constructive type theory*. Oxford Logic Guides, vol. 36. Oxford, United Kingdom: Clarendon Press.

Maurer, Luke, Downen, Paul, Ariola, Zena M., & Peyton Jones, Simon. (2017). Compiling without continuations. *Pages 482–494 of: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. PLDI '17. New York, NY, USA: ACM.

Mellies, Paul-Andre, & Vouillon, Jerome. (2005). Recursive polymorphic types and parametricity in an operational framework. *Pages 82–91 of: Proceedings of the 20th annual IEEE symposium on logic in computer science*. LICS '05. Washington, DC, USA: IEEE Computer Society.

Mendler, Nax P. (1988). *Inductive definition in type theory*. Ph.D. thesis, Cornell University.

Moggi, Eugenio. (1989). Computational lambda-calculus and monads. *Pages 14–23 of: Proceedings of the fourth annual symposium on logic in computer science*. Piscataway, NJ, USA: IEEE Press.

Munch-Maccagnoni, Guillaume. (2009). Focalisation and classical realisability. *Pages 409–423 of:* Grädel, Erich, & Kahle, Reinhard (eds), *Computer science logic: 23rd international workshop, CSL 2009, 18th annual conference of the EACSL*. CSL 2009. Berlin, Heidelberg: Springer Berlin Heidelberg.

Munch-Maccagnoni, Guillaume. (2013). *Syntax and models of a non-associative composition of programs and proofs*. Ph.D. thesis, Université Paris Diderot.

Munch-Maccagnoni, Guillaume. (2014). Formulae-as-types for an involutive negation. *Pages 70:1–70:10 of: Proceedings of the joint meeting of the twenty-third EACSL annual conference on computer science logic (CSL) and the twenty-ninth annual ACM/IEEE symposium on logic in computer science (LICS).* CSL-LICS '14. New York, NY, USA: ACM.

Munch-Maccagnoni, Guillaume, & Scherer, Gabriel. (2015). Polarised intermediate representation of lambda calculus with sums. *Pages 127–140 of: 2015 30th annual ACM/IEEE symposium on logic in computer science.* LICS 2015.

Ohori, Atsushi. (1999). The logical abstract machine: A Curry-Howard isomorphism for machine code. *Pages 300–318 of:* Middeldorp, Aart, & Sato, Taisuke (eds), *Functional and logic programming: 4th Fuji international symposium.* FLOPS '99. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ohori, Atsushi. (2003). Register allocation by proof transformation. *Pages 399–413 of:* Degano, Pierpaolo (ed), *Programming languages and systems: 12th European symposium on programming, ESOP 2003 held as part of the joint European conferences on theory and practice of software, ETAPS 2003.* ESOP 2003. Berlin, Heidelberg: Springer Berlin Heidelberg.

Oury, Nicolas. (2008). *Coinductive types and type preservation.* Message on the Coq-club mailing list.

Parigot, Michel. (1992). $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. *Pages 190–201 of:* Voronkov, Andrei (ed), *Logic programming and automated reasoning: International conference.* LPAR '92. Berlin, Heidelberg: Springer Berlin Heidelberg.

Peyton Jones, Simon, Tolmach, Andrew, & Hoare, Tony. (2001). Playing by the rules: Rewriting as a practical optimisation technique in GHC. *Haskell workshop.* ACM SIGPLAN.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, & Washburn, Geoffrey. (2006). Simple unification-based type inference for GADTs. *Pages 50–61 of: Proceedings of the eleventh ACM SIGPLAN international conference on functional programming.* ICFP '06. New York, NY, USA: ACM.

Peyton Jones, Simon L., & Launchbury, John. (1991). Unboxed values as first class citizens in a non-strict functional language. *Pages 636–666 of:* Hughes, John (ed), *Functional programming languages and computer architecture: 5th ACM conference.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Pfenning, Frank, & Davies, Rowan. (2001). A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, **11**(4), 511–540.

Pierce, Benjamin C. (2002). *Types and programming languages.* Cambridge, Massachusetts: The MIT Press.

Pitts, Andrew M. (1997). A note on logical relations between semantics and syntax. *Logic journal of IGPL*, **5**(4), 589–601.

Pitts, Andrew M. (2000). Parametric polymorphism and operational equivalence. *Mathematical structures in computer science*, **10**(3), 321–359.

Plotkin, Gordon D. (1975). Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, **1**, 125–159.

Polonovski, Emmanuel. (2004). *Explicit substitutions, logic and normalization.* Ph.D. thesis, Université Paris-Diderot - Paris VII.

Prawitz, Dag. (1974). On the idea of a general proof theory. *Synthese*, **27**(1/2), 63–77.

Pretnar, Matija. (2010). *The logic and handling of algebraic effects.* Ph.D. thesis, University of Edinburgh.

Reynolds, John C. (1974). Towards a theory of type structure. *Pages 408–423 of:* Robinet, Bernard (ed), *Programming symposium, proceedings colloque sur la programmation.* Lecture Notes in Computer Science, vol. 19. London, UK, UK: Springer-Verlag.

Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of:* Mason, R. E. A. (ed), *Proceedings of the IFIP 9th world computer congress.* Information Processing 83. Amsterdam: Elsevier Science Publishers B. V. (North-Holland).

Reynolds, John C. (1998). Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, **11**(4), 363–397.

Ronchi Della Rocca, Simona, & Paolini, Luca. (2004). *The parametric λ-calculus: a metamodel for computation.* Springer-Verlag.

Sabry, Amr, & Felleisen, Matthias. (1992). Reasoning about programs in continuation-passing style. *Pages 288–298 of: Proceedings of the 1992 ACM conference on LISP and functional programming.* LFP '92. New York, NY, USA: ACM.

Sabry, Amr, & Felleisen, Matthias. (1993). Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, **6**(3-4), 289–360.

Sabry, Amr, & Wadler, Philip. (1997). A reflection on call-by-value. *ACM transactions on programming languages and systems (TOPLAS)*, **19**(6), 916–941.

Schrijvers, Tom, Peyton Jones, Simon, Sulzmann, Martin, & Vytiniotis, Dimitrios. (2009). Complete and decidable type inference for GADTs. *Pages 341–352 of: Proceedings of the 14th ACM SIGPLAN international conference on functional programming*. ICFP '09. New York, NY, USA: ACM.

Selinger, Peter. (2001). Control categories and duality: On the categorical semantics of the lambda-mu calculus. *Mathematical structures in computer science*, **11**(2), 207–260.

Selinger, Peter. (2003). *Some remarks on control categories*. Unpublished Manuscript.

Singh, Satnam, Peyton Jones, Simon, Norell, Ulf, Pottier, François, Meijer, Erik, & McBride, Conor. (2011). *Sexy types — are we done yet?* Software Summit.

Turing, Alan M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, **42**(2), 230–265.

van Oostrom, Vincent. (1994). Confluence by decreasing diagrams. *Theoretical computer science*, **126**(2), 259–280.

Wadler, Philip. (2003). Call-by-value is dual to call-by-name. *Pages 189–201 of: Proceedings of the eighth ACM SIGPLAN international conference on functional programming*. New York, NY, USA: ACM.

Wadler, Philip. (2005). Call-by-value is dual to call-by-name, reloaded. *Pages 185–203 of:* Giesl, Jürgen (ed), *Term rewriting and applications: 16th international conference*. RTA 2005. Berlin, Heidelberg: Springer Berlin Heidelberg.

Wadler, Philip. (2015). Propositions as types. *Communications of the ACM*, **58**(12), 75–84.

Wright, Andrew K., & Felleisen, Matthias. (1994). A syntactic approach to type soundness. *Information and computation*, **115**(1), 38–94.

Zeilberger, Noam. (2008a). Focusing and higher-order abstract syntax. *Pages 359–369 of: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. POPL '08. New York, NY, USA: ACM.

Zeilberger, Noam. (2008b). On the unity of duality. *Annals of pure and applied logic*, **153**(1), 660–96.

Zeilberger, Noam. (2009). *The logical basis of evaluation order and pattern-matching*. Ph.D. thesis, Carnegie Mellon University.