Structures for Structural Recursion

Paul Downen Philip Johnson-Freyd Zena M. Ariola
University of Oregon, USA
{pdownen,philipjf,ariola}@cs.uoregon.edu

Abstract

Our goal is to develop co-induction from our understanding of induction, putting them on level ground as equal partners for reasoning about programs. We investigate several structures which represent well-founded forms of recursion in programs. These simple structures encapsulate reasoning by primitive and noetherian induction principles, and can be composed together to form complex recursion schemes for programs operating over a wide class of data and co-data types. At its heart, this study is guided by *duality*: each structure for recursion has a dual form, giving perfectly symmetric pairs of equal and opposite data and co-data types for representing recursion in programs. Duality is brought out through a framework presented in sequent style, which inherently includes control effects that are interpreted logically as classical reasoning principles. To accommodate the presence of effects, we give a calculus parameterized by a notion of strategy, which is strongly normalizing for a wide range of strategies. We also present a more traditional calculus for representing effect-free functional programs, but at the cost of losing some of the founding dualities.

Categories and Subject Descriptors F.3.3 [*Studies of Program Constructs*]: Program and recursion schemes

Keywords Recursion; Induction; Coinduction; Duality; Structures; Classical Logic; Sequent Calculus; Strong Normalization

1. Introduction

Martin-Löf's type theory [5, 15] 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 [17] 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 [19]:

> 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 . . . is coinduction and reasoning about corecursive 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 copatterns for coinduction [3] 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 [12]. Co-inductive co-data types are exciting because they may solve the existing problems with representing infinite objects in proof assistants like Coq [2].

The primary thrust of this work 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 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. Afterward, we 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 [16]), where induction allows for infinitely deep destruction and co-induction allows for infinitely deep construction. On the other hand, the copattern approach [2, 3] 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.

The primary calculus for recursion that we study corresponds to a classical logic, so it inherently contains *control effects* [11] that allow programs to abstract over their own control-flow—intuitionistic logic and effect-free functional programs are later considered as a special case. For that reason, 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 $length(\mathsf{Cons}\ (error\ \text{"}\mathtt{boom}\text{"})\ \mathsf{Nil})$ 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. Again, we disentangle this choice from the calculus itself, boiling down the distinction as a *discipline* for substitution. We get a family of calculi, parameterized by this substitution discipline, for reasoning about the behavior of programs ultimately executed with some evaluation strategy. The issue of strong normalization is then shown uniformly over this family of calculi by specifying some basic requirements of the chosen discipline.

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 to better highlight their symmetric opposition we base our language in 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 when considered programmatically: primitive recursion allows us to 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.

We begin by presenting some basic functional programs, including copatterns [3], in a sequent based syntax to illustrate how the sequent calculus gives a language for programming with structures and duality (Section 2) in which *all* types, including functions and polymorphism, are treated as user-defined data and co-data types (Section 3). Next, we develop 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 4). These two recursion schemes are incorporated into the sequent calculus language, and we demonstrate a rewriting theory that is strongly normalizing for well-typed programs and supports erasure of computationally irrelevant types at run-time (Section 5). Finally, we illustrate the natural deduction counterpart to our sequent calculus language, and show how the recursive constructs developed for classically effectful programs can be imported into a language for effect-free functional programming (Section 6).

2. 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 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 functional programs in this symmetric setting.

Example 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:

$$\begin{array}{ll}
\mathbf{data}\ \mathsf{Nat}\ \mathbf{where} & \mathbf{data}\ \mathsf{List}\ a\ \mathbf{where} \\
\quad \mathsf{Z} : \mathsf{Nat} & \quad \mathsf{Nil} : \mathsf{List}\ a \\
\quad \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat} & \quad \mathsf{Cons} : a \to \mathsf{List}\ a \to \mathsf{List}\ a
\end{array}$$

$$\begin{aligned}
&length\ \mathsf{Nil} &&= \mathsf{Z} \\
&length\ (\mathsf{Cons}\ x\ xs) &&= \mathbf{let}\ y = length\ xs\ \mathbf{in}\ \mathsf{S}\ y
\end{aligned}$$

This definition of $length$ describes its result for every possible call. Similarly, we can define $length$ in the $\mu\tilde{\mu}$-calculus[1] [8], a language based on Gentzen's sequent calculus, in much the same way. First, we introduce the types in question by data declarations in the sequent calculus:

$$\begin{array}{lrl|lrl}
\mathbf{data}\ \mathsf{Nat}\ \mathbf{where} & & & \mathbf{data}\ \mathsf{List}(a)\ \mathbf{where} & & \\
\quad \mathsf{Z} : & & \vdash \mathsf{Nat}\mid & \quad \mathsf{Nil} : & & \vdash \mathsf{List}(a)\mid \\
\quad \mathsf{S} : & \mathsf{Nat} & \vdash \mathsf{Nat}\mid & \quad \mathsf{Cons} : & a, \mathsf{List}(a) & \vdash \mathsf{List}(a)\mid
\end{array}$$

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 predefined 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 $\vdash \mathsf{List}(a)\mid$, 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 $\mathsf{Cons}(x, xs)$, much like in functional languages. Next, we define $length$ by specifying its behavior for every possible call:

$$\begin{aligned}
\langle length \| \mathsf{Nil} \cdot \alpha \rangle &= \langle \mathsf{Z} \| \alpha \rangle \\
\langle length \| \mathsf{Cons}(x, xs) \cdot \alpha \rangle &= \langle length \| xs \cdot \tilde{\mu} y. \langle \mathsf{S}(y) \| \alpha \rangle \rangle
\end{aligned}$$

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*, which connects together a producer and 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.

[1] Note that symbols μ and $\tilde{\mu}$ used here are not related to recursion, but rather are binders for variables and their dual co-variables in the tradition of [6].

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:

$$\begin{aligned} length\ \mathsf{Nil} &= \mathsf{Z} \\ length\ (\mathsf{Cons}\, x\ xs) &= \mathsf{S}(length\ xs) \end{aligned}$$

We can mimic this definition in the sequent calculus as:

$$\begin{aligned} \langle length \| \mathsf{Nil} \cdot \alpha \rangle &= \langle \mathsf{Z} \| \alpha \rangle \\ \langle length \| \mathsf{Cons}(x, xs) \cdot \alpha \rangle &= \langle \mathsf{S}(\mu\beta. \langle length \| xs \cdot \beta \rangle) \| \alpha \rangle \end{aligned}$$

Note that to represent the functional call $length\ xs$ inside the successor constructor S, we need to make use of a new kind of term: the output abstraction $\mu\beta.\langle length \| xs \cdot \beta\rangle$ names its output channel β before running the command $\langle length \| xs \cdot \beta \rangle$, which calls $length$ with xs as the argument and β as the return consumer. In the $\mu\tilde{\mu}$-calculus, 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. *End example* 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 dually 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*, 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 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:

$$\begin{aligned} &\mathbf{codata}\, \mathsf{Stream}(a)\, \mathbf{where} \\ &\quad \mathsf{Head} : \ \mid \mathsf{Stream}(a) \vdash a \\ &\quad \mathsf{Tail} : \ \mid \mathsf{Stream}(a) \vdash \mathsf{Stream}(a) \end{aligned}$$

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 $\mid \mathsf{Stream}(a) \vdash$. 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:

$$\begin{aligned} \langle countUp \| x \cdot \mathsf{Head}[\alpha] \rangle &= \langle x \| \alpha \rangle \\ \langle countUp \| x \cdot \mathsf{Tail}[\beta] \rangle &= \langle countUp \| \mathsf{S}(x) \cdot \beta \rangle \end{aligned}$$

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 its input data structure in its call-stack, whereas $countUp$ matches on its return co-data structure.

Abel *et al.* [3] have carried this intuition back into the functional paradigm. For example, we can still describe streams by their Head and Tail projections, and define $countUp$ through co-patterns:

$$\begin{aligned} &\mathbf{codata}\, \mathsf{Stream}\, a\, \mathbf{where} \\ &\quad \mathsf{Head} : \mathsf{Stream}\, a \to a \\ &\quad \mathsf{Tail} : \mathsf{Stream}\, a \to \mathsf{Stream}\, a \end{aligned}$$

$$\begin{aligned} (countUp\ x).\, \mathsf{Head} &= x \\ (countUp\ x).\, \mathsf{Tail} &= countUp\ (\mathsf{S}\, x) \end{aligned}$$

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 sequent calculus.

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 $\mathsf{SkipStream}(a) = \mathsf{Stream}(\mathsf{Maybe}(a))$, where $\mathsf{Maybe}(a)$ corresponds to the Haskell datatype with constructors $\mathsf{Nothing}$ and $\mathsf{Just}(x)$ for x of type a. 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 $\mathsf{S}^n(\mathsf{Z}), \mathsf{S}^{n-1}(\mathsf{Z}), \ldots, \mathsf{Z}, \mathsf{Nothing}, \ldots$:

$$\begin{aligned} \langle empty \| \mathsf{Head}[\alpha] \rangle &= \langle \mathsf{Nothing} \| \alpha \rangle \\ \langle empty \| \mathsf{Tail}[\beta] \rangle &= \langle empty \| \beta \rangle \end{aligned}$$

$$\begin{aligned} \langle countDown \| x \cdot \mathsf{Head}[\alpha] \rangle &= \langle \mathsf{Just}(x) \| \alpha \rangle \\ \langle countDown \| \mathsf{Z} \cdot \mathsf{Tail}[\beta] \rangle &= \langle empty \| \beta \rangle \\ \langle countDown \| \mathsf{S}(x) \cdot \mathsf{Tail}[\beta] \rangle &= \langle countDown \| x \cdot \beta \rangle \end{aligned}$$

End example 2.

Example 3. 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:

$$zeroes = \mathsf{Cons}\, \mathsf{Z}\, zeroes$$

We could emulate this definition in sequent style as the expansion of $zero$ when observed by any α:

$$\langle zeroes \| \alpha \rangle = \langle \mathsf{Cons}(\mathsf{Z}, zeroes) \| \alpha \rangle$$

Likewise, we can describe the concatenation of two, possibly infinite lists in the same way, by pattern-matching on the call:

$$\begin{aligned} \langle cat \| \mathsf{Nil} \cdot ys \cdot \alpha \rangle &= \langle ys \| \alpha \rangle \\ \langle cat \| \mathsf{Cons}(x, xs) \cdot ys \cdot \alpha \rangle &= \langle \mathsf{Cons}(x, \mu\beta. \langle cat \| xs \cdot ys \cdot \beta \rangle) \| \alpha \rangle \end{aligned}$$

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* 3.

3. A Higher-Order Sequent Calculus

Based on our example programs in Section 2, we now flesh out more formally a higher-order language of the sequent calculus: the $\mu\tilde{\mu}$-calculus. The full syntax of this language is shown in Figure 1. The different components of programs in the $\mu\tilde{\mu}$-calculus can be understood by their relationship between opposing forces of input and output. A term, v, produces an output, a co-term, e, consumes

[2] 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.

$$c \in Command ::= \langle v \| e \rangle$$
$$v \in Term ::= x \mid \mu\alpha.c \mid \mathsf{K}(\vec{e}, \vec{v}) \mid \mu(\mathsf{H}[\vec{x}, \vec{\alpha}].c \mid \ldots) \qquad e \in CoTerm ::= \alpha \mid \tilde{\mu}x.c \mid \tilde{\mu}[\mathsf{K}(\vec{\alpha}, \vec{x}).c \mid \ldots] \mid \mathsf{H}[\vec{v}, \vec{e}]$$
$$A, B, C, D \in Type ::= a \mid \lambda a : k.B \mid A\ B \mid \mathsf{F}(\vec{A}) \mid \mathsf{G}(\vec{A}) \qquad k, l \in Kind ::= \star \mid k_1 \to k_2$$

Figure 1. The syntax of the higher-order $\mu\tilde{\mu}$-calculus.

an input, and a command, c, neither produces nor consumes, it just *runs*. Thus, we can consider commands to be the computational unit of the language: when we talk about running a program, it is a command which does the running, not a term.

To begin, we focus on the *core* of the $\mu\tilde{\mu}$-calculus, which includes just the substrate necessary for piping inputs and outputs to the appropriate places. In particular, we have two different forms of inputs and outputs: the implicit, unnamed inputs and outputs of terms and co-terms, and the explicit, named inputs and outputs introduced by variables (typically written x, y, z) and co-variables (typically written α, β, γ). Thus, besides variables and co-variables, the core $\mu\tilde{\mu}$-calculus includes the generic abstractions seen in Section 2, $\mu\alpha.c$ and $\tilde{\mu}x.c$, which mediate between named and unnamed inputs and outputs. The output of the term $\mu\alpha.c$ is named α in the command c, and dually the input of the co-term $\tilde{\mu}x.c$ is named x in c.

Even though the core $\mu\tilde{\mu}$-calculus has not introduced any specific types yet, we can still consider its type system for ensuring proper communication between producers and consumers, shown in Figure 2. The (typed) free variables and co-variables are tracked in separate contexts, written Γ and Δ respectively, and the entailment ($\vdash$) separates inputs on the left from outputs on the right. Additionally, the context, Θ, for type variables (written a, b, c, d), being neither input nor output, adorn the turnstyle itself. Since programs of the $\mu\tilde{\mu}$-calculus are made up of three different forms of components, the typing rules use three different forms of sequents: $\Gamma \vdash_\Theta v : A | \Delta$ states that v is a term producing an output of type A, $\Gamma | e : A \vdash_\Theta \Delta$ states that e is a co-term consuming an input of type A, and $c : (\Gamma \vdash_\Theta \Delta)$ states that c is a well-typed command. The language of types and kinds is just the simply typed λ-calculus at the type level with $\star$ as the base kind, $\Theta \vdash A : k$ states that A is a type of kind k, and $\Theta \vdash A = B : k$ states that A and B are $\alpha\beta\eta$-equivalent types of kind k.

This core language does not include any baked-in types. Instead, all types are user-defined by a general declaration mechanism for (co-)data types introduced in [8], similar to the data declaration mechanisms of functional languages but generalized through duality. Data declarations introduce new constructed terms as well as a new *case abstraction* co-term that performs case analysis to destruct its input before deciding which branch to take similar to the context $\mathbf{case} \square \mathbf{of} \ldots$ in functional languages. Co-data declarations are exactly symmetric, introducing new constructed co-terms as well as a new case abstraction term that performs case analysis on its output before deciding how to respond.

We already saw some example declarations previously for Nat, $\mathsf{List}(a)$, and $\mathsf{Stream}(a)$. As it turns out, *all* the basic types from functional programming languages follow the same pattern and can be declared as user-defined types. For example, pairs are defined as:

$$\mathbf{data}\,(a:\star) \otimes (b:\star)\,\mathbf{where}$$
$$(\_,\_) : \quad a, b \vdash a \otimes b |$$

which says that building a pair of type $a \otimes b$ requires the terms v of type a and v' of type b, obtaining the constructed pair (v, v'). Destruction of pairs, expressed by the case abstraction co-term $\tilde{\mu}[(x, y).c]$, pattern-matches on its input pair before running the command c.

Furthermore, we can declare the type for functions as:

$$\mathbf{codata}\,(a:\star) \to (b:\star)\,\mathbf{where}$$
$$\_\cdot\_ : \quad a | a \to b \vdash b$$

This co-data declaration says that building a function call-stack of type $a \to b$ requires a term v of type a and a co-term e of type b, obtaining the constructed stack $v \cdot e$. Destruction of call-stacks, expressed by the case abstraction term $\mu(x \cdot \alpha.c)$, pattern-matches on its output stack before running c. Note that this is an alternative representation of functions to λ-abstractions in functional languages, but an equivalent one. Indeed, the two views of functions are mutually definable:

$$\lambda x.v = \mu(x \cdot \alpha.\langle v \| \alpha \rangle) \qquad \mu(x \cdot \alpha.c) = \lambda x.\mu\alpha.c$$

Here, we generalize the declaration mechanism from [8] to include higher-order types and quantified type variables. The general forms of (non-recursive) data and co-data declaration in the $\mu\tilde{\mu}$-calculus are given in Figure 3, and the associated typing rules in Figure 4. In addition to the rule for determining when the (co-)data types $\mathsf{F}(\vec{A})$ and $\mathsf{G}(\vec{A})$ are well-kinded, we also have the left and right rules for typing (co-)data structures and case abstractions. By instantiating the (co-)data type constructors at the types $\vec{A}$, we must substitute $\vec{A}$ for all possible occurrences of the parameters $\vec{a}$ in the declaration. Furthermore, the chosen instances $\vec{D}$ for the quantified type variables $\vec{d_i}$, which annotate the constructor, must also be substituted for their occurrences in other types. With this in mind, the rules for construction (the $\mathsf{F}R\mathsf{K}_i$ and $\mathsf{G}L\mathsf{H}_i$ rules) check that the sub-(co-)terms and quantified types of a structure have the expected instantiated types, whereas the rules for deconstruction ($\mathsf{F}L$ and $\mathsf{G}R$) extend the typing contexts with the appropriately typed (co-)variables and type variables.

This form of (co-)data type declaration lets us express not only existential quantification—as in Haskell and Coq—but also universal quantification as well:

$$\mathbf{data}\,\exists(a:\star\to\star)\,\mathbf{where} \qquad \mathbf{codata}\,\forall(a:\star\to\star)\,\mathbf{where}$$
$$\mathsf{Pack} : \quad a\ b \vdash_{b:\star} \exists a | \qquad \mathsf{Spec} : \quad |\forall a \vdash_{b:\star} a\ b$$

Notice that these general patterns give us the expected typing rules:

$$\frac{c : \Gamma \vdash_{\Theta, b:\star} \alpha : A\ b, \Delta}{\Gamma \vdash_\Theta \mu(\mathsf{Spec}^{b:\star}[\alpha].c) : \forall A | \Delta} \qquad \frac{\Theta \vdash B : \star \quad \Gamma | e : A\ B \vdash_\Theta \Delta}{\Gamma | \mathsf{Spec}^B[e] : \forall A \vdash_\Theta \Delta}$$

$$\frac{\Theta \vdash B : \star \quad \Gamma \vdash_\Theta v : A\ B | \Delta}{\Gamma \vdash_\Theta \mathsf{Pack}^B(v) : \exists A | \Delta} \qquad \frac{c : \Gamma, x : A\ b \vdash_{\Theta, b:\star} \Delta}{\Gamma | \tilde{\mu}[\mathsf{Pack}^{b:\star}(x).c] : \exists A \vdash_\Theta \Delta}$$

Using a recursively-defined case abstraction with deep pattern-matching, we can now represent $length$ in the $\mu\tilde{\mu}$-calculus:

$$\begin{aligned} length = \mu(&\mathsf{Nil} \cdot \alpha.\langle \mathsf{Z} \| \alpha \rangle \\ &|\mathsf{Cons}(x, xs) \cdot \alpha.\langle length \| xs \cdot \tilde{\mu}y.\langle \mathsf{S}(y) \| \alpha \rangle \rangle) \end{aligned}$$

Furthermore, the deep pattern-matching can be mechanically translated to the shallow case analysis for (co-)data types:

$$\begin{aligned} length = \mu(xs \cdot \alpha.\ \langle xs \| \tilde{\mu}[&\mathsf{Nil}.\langle \mathsf{Z} \| \alpha \rangle \\ &|\mathsf{Cons}(x, xs').\langle length \| xs' \cdot \tilde{\mu}y.\langle \mathsf{S}(y) \| \alpha \rangle \rangle] \rangle) \end{aligned}$$

This case abstraction describes exactly the same specification as the definition for $length$ in Example 1.

$$\frac{}{\Gamma, x : A \vdash_\Theta x : A|\Delta}\ Var \qquad \frac{c : (\Gamma \vdash_\Theta \alpha : A, \Delta)}{\Gamma \vdash_\Theta \mu\alpha.c : A|\Delta}\ Act \qquad \frac{c : (\Gamma, x : A \vdash_\Theta \Delta)}{\Gamma|\tilde{\mu}x.c : A \vdash_\Theta \Delta}\ CoAct \qquad \frac{}{\Gamma|\alpha : A \vdash_\Theta \alpha : A, \Delta}\ CoVar$$

$$\frac{\Theta \vdash A = B : \star \quad \Gamma \vdash_\Theta v : A|\Delta}{\Gamma \vdash_\Theta v : B|\Delta}\ Eq \qquad \frac{\Gamma \vdash_\Theta v : A|\Delta \quad \Theta \vdash A : \star \quad \Gamma|e : A \vdash_\Theta \Delta}{\langle v\|e\rangle : (\Gamma \vdash_\Theta \Delta)}\ Cut \qquad \frac{\Theta \vdash A = B : \star \quad \Gamma|e : B \vdash_\Theta \Delta}{\Gamma|e : A \vdash_\Theta \Delta}\ CoEq$$

Figure 2. The type system for the core higher-order $\mu\tilde{\mu}$-calculus.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\, \mathsf{F}(\overrightarrow{a:k})\, \mathbf{where}} \\
\mathsf{K}_1 : & \overrightarrow{B_1} \vdash_{\overrightarrow{d_1 : l_1}} \mathsf{F}(\vec{a})|\overrightarrow{C_1} \\
\quad\ldots & \\
\mathsf{K}_n : & \overrightarrow{B_n} \vdash_{\overrightarrow{d_n : l_n}} \mathsf{F}(\vec{a})|\overrightarrow{C_n}
\end{array}
\qquad\qquad
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{codata}\, \mathsf{G}(\overrightarrow{a:k})\, \mathbf{where}} \\
\mathsf{H}_1 : & \overrightarrow{B_1}\,|\, \mathsf{G}(\vec{a}) \vdash_{\overrightarrow{d_1 : l_1}} \overrightarrow{C_1} \\
\quad\ldots & \\
\mathsf{H}_n : & \overrightarrow{B_n}\,|\, \mathsf{G}(\vec{a}) \vdash_{\overrightarrow{d_n : l_n}} \overrightarrow{C_n}
\end{array}$$

Figure 3. General form of declarations for user-defined data and co-data.

$$\frac{\overrightarrow{\Theta \vdash A : k}}{\Theta \vdash \mathsf{F}(\vec{A}) : \star} \qquad \frac{c_1 : (\Gamma, \overrightarrow{x : B_1\{\overrightarrow{A/a}\}} \vdash_{\Theta, \overrightarrow{d_1 : l_1\{\overrightarrow{A/a}\}}} \Delta, \overrightarrow{\alpha : C_1\{\overrightarrow{A/a}\}}) \quad \ldots}{\Gamma|\tilde{\mu}[\mathsf{K}_1^{\overrightarrow{d_1 : l_1}}(\vec{\alpha}, \vec{x}).c_1|\ldots] : \mathsf{F}(\vec{A}) \vdash_\Theta \Delta}\ \mathsf{F}L$$

$$\frac{\overrightarrow{\Theta \vdash D : l_i\{\overrightarrow{A/a}\}} \quad \overrightarrow{\Gamma|e : C_i\{\overrightarrow{A/a}, \overrightarrow{D/d_i}\} \vdash_\Theta \Delta} \quad \overrightarrow{\Gamma \vdash_\Theta v : B_i\{\overrightarrow{A/a}, \overrightarrow{D/d_i}\}|\Delta}}{\Gamma \vdash_\Theta \mathsf{K}_i^{\vec{D}}(\vec{e}, \vec{v}) : \mathsf{F}(\vec{A})|\Delta}\ \mathsf{F}R\mathsf{K}_i$$

$$\frac{\overrightarrow{\Theta \vdash A : k}}{\Theta \vdash \mathsf{G}(\vec{A}) : \star} \qquad \frac{c_1 : (\Gamma, x : B_1\{\overrightarrow{A/a}\} \vdash_{\Theta, \overrightarrow{d_1 : l_1\{\overrightarrow{A/a}\}}} \alpha : C_1\{\overrightarrow{A/a}\}, \Delta) \quad \ldots}{\Gamma \vdash_\Theta \mu(\mathsf{H}_1^{\overrightarrow{d_1 : l_1}}[\vec{x}, \vec{\alpha}].c_1|\ldots) : \mathsf{G}(\vec{A})|\Delta}\ \mathsf{G}R$$

$$\frac{\overrightarrow{\Theta \vdash D : l_i\{\overrightarrow{A/a}\}} \quad \overrightarrow{\Gamma \vdash_\Theta v : B_i\{\overrightarrow{A/a}, \overrightarrow{D/d}\}|\Delta} \quad \overrightarrow{\Gamma|e : C_i\{\overrightarrow{A/a}, \overrightarrow{D/d}\} \vdash_\Theta \Delta}}{\Gamma|\, \mathsf{H}_i^{\vec{D}}[\vec{v}, \vec{e}] : \mathsf{G}(\vec{A}) \vdash_\Theta \Delta}\ \mathsf{G}L\mathsf{H}_i$$

Figure 4. Typing rules for non-recursive, user-defined data and co-data types.

In each of the examples in Section 2, 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 1 and 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 3 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 $\mu\tilde{\mu}$-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 5).

4. Well-Founded Recursion

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 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 $\mathsf{S}(length\ xs)$. 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 [1]. 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, 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.

4.1 Primitive Recursion

We begin with the 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.

Let's consider some examples of using natural number indices for the purpose of defining (co-)data types with recursive structures. We extend the (co-)type declaration mechanism seen previously 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—for the zero and successor cases—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:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathbf{data}\,\mathsf{Vec}(i : \mathsf{Ix}, a : \star)\ \mathbf{by}\ \text{primitive recursion on } i} \\ \mathbf{where}\, i = 0 & \mathsf{Nil} : & \vdash \mathsf{Vec}(0, a)| \\ \mathbf{where}\, i = j + 1 & \mathsf{Cons} : & a, \mathsf{Vec}(j, a) \vdash \mathsf{Vec}(j+1, a)| \end{array}$$

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 $\mathsf{Vec}(N + 1, A)$. 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.[3] 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:

$$\frac{c : (\Gamma \vdash_\Theta \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Nil}.c] : \mathsf{Vec}(0, A) \vdash_\Theta \Delta}\ \mathsf{Vec}\, L_0$$

$$\frac{c : (\Gamma, x : A, xs : \mathsf{Vec}(M, A) \vdash_\Theta \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Cons}(x, xs).c] : \mathsf{Vec}(M+1, A) \vdash_\Theta \Delta}\ \mathsf{Vec}\, L_{+1}$$

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:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathbf{data}\,\mathsf{IxList}(i : \mathsf{Ix}, a : \star)\ \mathbf{by}\ \text{primitive recursion on } i} \\ \mathbf{where}\, i = 0 & \mathsf{Nil} : & \vdash \mathsf{IxList}(0, a)| \\ \mathbf{where}\, i = j + 1 & \mathsf{Nil} : & \vdash \mathsf{IxList}(j+1, a)| \\ & \mathsf{Cons} : & a, \mathsf{IxList}(j, a) \vdash \mathsf{IxList}(j+1, a)| \end{array}$$

Now, destructing a non-zero $\mathsf{IxList}(N + 1, A)$ requires both cases, as given in the co-term $\tilde{\mu}[\mathsf{Nil}.c|\mathsf{Cons}(x, xs).c']$. IxList has three right rules for building terms: for Nil at both 0 and $M + 1$ and for Cons. It also has two left rules: one for case abstractions handling the constructors of the 0 case and another for the $M + 1$ case.

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:

$$P[0] \land (\forall j : \mathbb{N}.P[j] \to P[j+1]) \to (\forall i : \mathbb{N}.P[i])$$

and likewise the refutation of such a statement, as is given by any specific counter-example—$n : \mathbb{N} \land \overline{P[n]} \to \overline{(\forall i : \mathbb{N}.P[i])}$—into logical rules of the sequent calculus.[4] By the usual reading of sequents, 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 with a type named $\mathsf{Inflate}$, so that the primitive recursive proposition $\forall i : \mathbb{N}.P[i]$ is written as the type $\mathsf{Inflate}(\lambda i : \mathsf{Ix}.A)$ with the inference rules:

$$\frac{\vdash A\,0 \quad A\,j \vdash_{j:\mathsf{Ix}} A\,(j+1)}{\vdash \mathsf{Inflate}(A)} \qquad \frac{\vdash M : \mathsf{Ix} \qquad A\,M \vdash}{\mathsf{Inflate}(A) \vdash}$$

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 declaration for $\mathsf{Inflate}$:

$$\begin{array}{l} \mathbf{codata}\,\mathsf{Inflate}(a : \mathsf{Ix} \to \star)\ \mathbf{where} \\ \mathsf{Up} : \quad | \,\mathsf{Inflate}(a) \vdash_{j:\mathsf{Ix}} a\ j \end{array}$$

The general mechanism for co-data automatically generates the 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 $\mathsf{Inflate}$, 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 $\mu(\mathsf{Up}^{0:\mathsf{Ix}}[\alpha].c_0|\,\mathsf{Up}^{j+1:\mathsf{Ix}}[\alpha](x).c_1)$ 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 $\mathsf{Up}^{j+1:\mathsf{Ix}}[\alpha](x)$. This term has the typing rule:

$$\frac{c_0 : (\Gamma \vdash_\Theta \alpha : A\,0, \Delta) \quad c_1 : (\Gamma, x : A\,j \vdash_{\Theta, j:\mathsf{Ix}} \alpha : A\,(j+1), \Delta)}{\Gamma \vdash_\Theta \mu(\mathsf{Up}^{0:\mathsf{Ix}}[\alpha].c_0|\,\mathsf{Up}^{j+1:\mathsf{Ix}}[\alpha](x).c_1) : \mathsf{Inflate}(A)|\Delta}$$

Terms of type $\mathsf{Inflate}\, i : \mathsf{Ix}\,.A$ (which is shorthand for the type $\mathsf{Inflate}(\lambda i : \mathsf{Ix}.A)$) describe a process which is able to produce $A\{N/i\}$, for any index N, by stepwise producing $A\{0/i\}$, $A\{1/i\}$, $\ldots$, $A\{N/i\}$ and piping the previous output to the recursive input

[3] 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 $\mathsf{Vec}(i, A)$.

[4] 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.

x of the next step, thus "inflating" the index in the result arbitrarily high. The index of the particular step being handled is part of the constructor pattern, so that the recursive case abstraction knows which branch to take. In contrast, co-terms of type $\mathsf{Inflate}\, i : \mathsf{Ix}\,.A$ *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 Deflate:

$$\begin{array}{l}\textbf{data}\ \mathsf{Delfate}(a : \mathsf{Ix} \to \star)\ \textbf{where}\\ \quad \mathsf{Down} : \quad a\ j \vdash_{j:\mathsf{Ix}} \mathsf{Deflate}(a)|\end{array}$$

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:

$$\frac{c_1 : (\Gamma, x : A\ (j+1) \vdash_{\Theta, j:\mathsf{Ix}} \alpha : A\ j, \Delta) \quad c_0 : (\Gamma, x : A\ 0 \vdash_\Theta \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Down}^{0:\mathsf{Ix}}(x).c_0|\mathsf{Down}^{j+1:\mathsf{Ix}}(x)[\alpha].c_1] : \mathsf{Deflate}(A) \vdash_\Theta \Delta}$$

Dual to before, the recursive output sink can be accessed through the co-variable α in the pattern $\mathsf{Down}^{j+1:\mathsf{Ix}}(x)[\alpha]$. The terms of type $\mathsf{Deflate}\, i : \mathsf{Ix}\,.A$ hide the particular index at which they produce an output. In contrast, it is now the co-terms of the type $\mathsf{Deflate}\, i : \mathsf{Ix}\,.A$ which describe a process which is able to consume $A\{N/i\}$ for any choice of N in steps by consuming $A\{N/i\}, \dots, A\{0/i\}$ and piping the previous input to the recursive output α of the next step, thus "deflating" the index in the input down to 0.

4.2 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 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:

$$\begin{array}{l}\textbf{data}\ \mathsf{Nat}(i : \mathsf{Ord})\ \textbf{by}\ \text{noetherian recursion on } i\ \textbf{where}\\ \quad \mathsf{Z} : \qquad\quad \vdash \mathsf{Nat}(i)|\\ \quad \mathsf{S} : \quad \mathsf{Nat}(j) \vdash_{j<i} \mathsf{Nat}(i)|\end{array}$$

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:

$$\frac{}{\Gamma \vdash_\Theta \mathsf{Z} : \mathsf{Nat}(N)|\Delta}\ \mathsf{Nat}R\,\mathsf{Z}$$

$$\frac{\Theta \vdash M < N \quad \Gamma \vdash_\Theta v : \mathsf{Nat}(M)|\Delta}{\Gamma \vdash_\Theta \mathsf{S}^M(v) : \mathsf{Nat}(N)|\Delta}\ \mathsf{Nat}R\,\mathsf{S}$$

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:

$$\frac{c_0 : (\Gamma \vdash_\Theta \Delta) \quad c_1 : (\Gamma, x : \mathsf{Nat}(j) \vdash_{\Theta, j<N} \Delta)}{\Gamma|\tilde{\mu}[\mathsf{Z}.c_0|\mathsf{S}^{j<N}(x).c_1] : \mathsf{Nat}(N) \vdash_\Theta \Delta}\ \mathsf{Nat}L$$

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 1:

$$\begin{array}{l}\textbf{data}\ \mathsf{List}(i : \mathsf{Ord}, a : \star)\ \textbf{by}\ \text{noetherian recursion on } i\ \textbf{where}\\ \quad \mathsf{Nil} : \qquad\qquad\qquad \vdash \mathsf{List}(i,a)|\\ \quad \mathsf{Cons} : \quad a, \mathsf{List}(j,a) \vdash_{j<i} \mathsf{List}(i,a)|\end{array}$$

Furthermore, the infinite streams from Example 2 can also be defined as a co-data type by noetherian recursion:

$$\begin{array}{l}\textbf{codata}\ \mathsf{Stream}(i : \mathsf{Ord}, a : \star)\ \textbf{by}\ \text{noetherian recursion on } i\ \textbf{where}\\ \quad \mathsf{Head} : \quad |\,\mathsf{Stream}(i,a) \vdash a\\ \quad \mathsf{Tail} : \quad |\,\mathsf{Stream}(i,a) \vdash_{j<i} \mathsf{Stream}(j,a)\end{array}$$

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 $M < N$:

$$\frac{\Gamma|e : A \vdash_\Theta \Delta}{\Gamma|\,\mathsf{Head}[e] : \mathsf{Stream}(N,A) \vdash_\Theta \Delta}\ \mathsf{Stream}L\,\mathsf{Head}$$

$$\frac{\Theta \vdash M < N \quad \Gamma|e : \mathsf{Stream}(M,A) \vdash_\Theta \Delta}{\Gamma|\,\mathsf{Tail}^M[e] : \mathsf{Stream}(N,A) \vdash_\Theta \Delta}\ \mathsf{Stream}L\,\mathsf{Tail}$$

Whereas a $\mathsf{Stream}(N, A)$ value is given by pattern-matching on these two possible observations:

$$\frac{c : (\Gamma \vdash_\Theta \alpha : A, \Delta) \quad c' : (\Gamma \vdash_{\Theta, j<N} \beta : \mathsf{Stream}(j,A), \Delta)}{\Gamma|\mu(\mathsf{Head}[\alpha].c|\mathsf{Tail}^{j<N}[\beta].c') : \mathsf{Stream}(N,A) \vdash_\Theta \Delta}\ \mathsf{Stream}R$$

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 counterexample within the chosen bound. 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 Ascend:

$$\frac{\mathsf{Ascend}(j,A) \vdash_{j<N} A\ j}{\vdash \mathsf{Ascend}(N,A)} \qquad \frac{\vdash M < N \quad A\ M \vdash}{\mathsf{Ascend}(N,A) \vdash}$$

Note that we will write $\mathsf{Ascend}\, i < N.A$ as shorthand for the type $\mathsf{Ascend}(N, \lambda i : \mathsf{Ord}.A)$. 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 Ascend as a co-data type of the form:

$$\begin{array}{l} \mathbf{codata}\, \mathsf{Ascend}(i : \mathsf{Ord}, a : \mathsf{Ord} \to \star)\, \mathbf{where} \\ \quad \mathsf{Rise} : \quad \mid \mathsf{Ascend}(i, a) \vdash_{j<i} a\ j \end{array}$$

Again, the general mechanism for co-data types tells us how to construct the counter-example with Rise, and destruct it by simple case analysis. The recursive form of case analysis is given manually as the term $\mu(\mathsf{Rise}^{j<N}[\alpha](x).c)$, 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:

$$\frac{c : (\Gamma, x : \mathsf{Ascend}(j, A) \vdash_{\Theta, j<N} \alpha : A\ j, \Delta)}{\Gamma \vdash_\Theta \mu(\mathsf{Rise}^{j<N}[\alpha](x).c) : \mathsf{Ascend}(N, A) | \Delta}$$

In essence, the terms of type $\mathsf{Ascend}\, i < N.A$ describe a process which is capable of producing $A\{M/i\}$ for any $M < N$ by leaps and bounds: an output of type $A\{M/i\}$ 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 $\mathsf{Ascend}\, i < N.A$ 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 Descend, with the following data declaration and additional typing rule for recursive case analysis:

$$\begin{array}{l} \mathbf{data}\, \mathsf{Descend}(i : \mathsf{Ord}, a : \mathsf{Ord} \to \star)\, \mathbf{where} \\ \quad \mathsf{Fall} : \quad a\ j \vdash_{j<i} \mathsf{Descend}(i, a) | \end{array}$$

$$\frac{c : (\Gamma, x : A\ j \vdash_{\Theta, j<N} \alpha : \mathsf{Descend}(j, A), \Delta)}{\Gamma | \tilde{\mu}[\mathsf{Fall}^{j<N}(x)[\alpha].c] : \mathsf{Descend}(N, A) \vdash_\Theta \Delta}$$

Now that the roles are reversed, the terms of $\mathsf{Descend}\, i < N.A$ hide the chosen index M at which they can produce a result of type $A\{M/i\}$. Instead, the co-terms of $\mathsf{Descend}\, i < N.A$ consuming $A\{M/i\}$ for any index $M < N$: an input of type $A\{M/i\}$ is broken down by repeating the same process whenever it is necessary to descend from an index under M.

5. A Parametric Sequent calculus with Recursion

We now flesh out the rest of the system for recursive types and structures for representing recursive programs in the sequent calculus. The core rules for kinding and sorting, which accounts for both forms of type-level indices, are given in Figure 5. 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 kind $\star$) 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. Thus, this sorting system categorizes the distinction between erasable and non-erasable type annotations found in programs.

Before admitting a user-defined (co-)data type into the system, we need to check that its declaration actually denotes a meaningful type. For the non-recursive (co-)data declarations, like those in Figure 3, this well-formedness check just confirms that the sequent associated to each constructor K_i or H_i is well-formed, given by a derivation of $(\overrightarrow{B_i} \vdash_{\overrightarrow{a:k}, \overrightarrow{d_i:l_i}} \overrightarrow{C_i})\, \mathbf{seq}$ from Figure 5. When checking for well-formedness of (co-)data types defined by primitive induction on $i : \mathsf{Ix}$, as with the general form

$$\begin{array}{llll} \multicolumn{4}{l}{\mathbf{data}\, \mathsf{F}(i : \mathsf{Ix}, \overrightarrow{a : k})\, \mathbf{by}\ \text{primitive recursion on}\ i} \\ \mathbf{where}\, i = 0 & \mathsf{K}_1 : & \overrightarrow{B_1} \vdash_{\overrightarrow{d_1:l_1}} \mathsf{F}(0, \vec{a}) | \overrightarrow{C_1} & \ldots \\ \mathbf{where}\, i = j + 1 & \mathsf{K}'_1 : & \overrightarrow{B'_1} \vdash_{\overrightarrow{d'_1:l'_1}} \mathsf{F}(j + 1, \vec{a}) | \overrightarrow{C'_1} & \ldots \end{array}$$

the $i = 0$ case proceeds by checking that the sequents are well-formed for each constructor $K_1 \ldots$ without referencing i, $(\overrightarrow{B_1} \vdash_{\overrightarrow{a:k}, \overrightarrow{d_1:l_1}} \overrightarrow{C_1})\, \mathbf{seq}$, and in the $i = j + 1$ case we check each $(\overrightarrow{B'_1} \vdash_{j:\mathsf{Ix}, \overrightarrow{a:k}, \overrightarrow{d'_1:l'_1}} \overrightarrow{C'_1})\, \mathbf{seq}$ with the extra rule

$$\frac{\overrightarrow{\Theta, j : \mathsf{Ix}, \Theta' \vdash A : k}}{\Theta, j : \mathsf{Ix}, \Theta' \vdash \mathsf{F}(j, \vec{A}) : \star}$$

Intuitively, in the $i = j + 1$ case the sequents for the constructors may additionally refer to smaller instances $\mathsf{F}(j, \vec{A})$ of the type being defined. If the declaration is well-formed, we add the typing rules for F similarly to a non-recursive (co-)data type. The difference is that the constructors for the $i = 0$ and $i = j + 1$ case build a structure of type $\mathsf{F}(0, \vec{A})$ and $\mathsf{F}(M + 1, \vec{A})$ with M substituted for j, respectively. Additionally, there are two case abstractions: one of type $\mathsf{F}(0, \vec{A})$ 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 $\mathsf{F}(i : \mathsf{Ord}, \overrightarrow{a : k})$ defined by noetherian induction on $i : \mathsf{Ord}$, we get to assume the type is defined for smaller indices:

$$\frac{\Theta, i : \mathsf{Ord}, \Theta' \vdash M < i \quad \overrightarrow{\Theta, i : \mathsf{Ord}, \Theta' \vdash A : k}}{\Theta, i : \mathsf{Ord}, \Theta' \vdash \mathsf{F}(M, \vec{A}) : \star}$$

Intuitively, the sequents for the constructors may refer to $\mathsf{F}(M, \vec{A})$, so long as they introduce quantified type variables $\overrightarrow{d : l}$ such that $\overrightarrow{a : k}, \overrightarrow{d : l} \vdash M < i$. Other than this, the typing rules for structures and case statements are exactly the same as for non-recursive (co-)data types.

We also give the rewriting theory for the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus in Figure 6, which is parameterized by the strategy $\mathcal{S}$. Since the classical sequent calculus inherently admits control effects, the result of a program can completely change depending on the strategy—$\langle length \| \mathsf{Rise}^2[\mathsf{Cons}(\mu\delta.\langle 13 \| \alpha\rangle, \mathsf{Nil}), \alpha]\rangle$ results in $\langle 1 \| \alpha\rangle$ under call-by-name evaluation and $\langle 13 \| \alpha\rangle$ under call-by-value—so that the parametric $\mu\tilde{\mu}_{\mathcal{S}}$-calculus is actually a family of related but different rewriting theories for reasoning about different evaluation strategies, thus enabling strategy-independent reasoning. The choice of strategy is given as the syntactic notions of *value* and *co-value*: $\mathcal{S}$ is the subset of terms $V \in Value$ and $E \in CoValue$ which may be substituted for (co-)variables. In other words, the strategy refines the range of significance for (co-)variables by limiting what they might stand in for, and in this way it resolves the conflict between both the μ- and $\tilde{\mu}$-abstractions [6]. For example, the strategies for call-by-value and call-by-name evaluation are shown in Figure 8, and a strategy respresenting call-by-need evaluation is representable this way as well [8].

The reduction rules are derived from the core theory of substitution in $\mu\tilde{\mu}_{\mathcal{S}}$ (the top rules of Figure 6), plus rules derived from generic β and η principles for every (co-)data type. Of note are the ς rules, first appearing in Wadler's dual calculus [20], and which we derive from the $\beta\eta$ principles for any (co-)data type [8]. The general lifting rules for (co-)data types are described by the lifting contexts

$$\frac{}{\Theta \vdash 0 : \mathsf{Ix}} \qquad \frac{\Theta \vdash M : \mathsf{Ix}}{\Theta \vdash M+1 : \mathsf{Ix}} \qquad \frac{}{\Theta \vdash 0 < \infty} \qquad \frac{\Theta \vdash M < \infty}{\Theta \vdash M+1 < \infty} \qquad \frac{\Theta \vdash M < N}{\Theta \vdash M < M+1} \qquad \frac{\Theta \vdash M < N \quad \Theta \vdash N < N'}{\Theta \vdash M < N'}$$

$$\frac{a : k \notin \Theta'}{\Theta, a : k, \Theta' \vdash a : k} \qquad \frac{\Theta, a : k_1 \vdash B : k_2 \quad \Theta \vdash k_2 : \square}{\Theta \vdash \lambda a : k.B : k_1 \to k_2} \qquad \frac{\Theta \vdash A : k_1 \to k_2 \quad \Theta \vdash B : k_1}{\Theta \vdash A\, B : k_2} \qquad \frac{\Theta \vdash M < N \quad \Theta \vdash N : \mathsf{Ord}}{\Theta \vdash M : \mathsf{Ord}} \qquad \frac{}{\Theta \vdash \infty : \mathsf{Ord}}$$

$$\frac{\Theta \vdash k : \square}{\Theta \vdash k : \blacksquare} \qquad \frac{\Theta \vdash k_1 : \blacksquare \quad \Theta \vdash k_2 : \square}{\Theta \vdash k_1 \to k_2 : \square} \qquad \frac{}{\Theta \vdash \star : \square} \qquad \frac{}{\Theta \vdash \mathsf{Ix} : \blacksquare} \qquad \frac{}{\Theta \vdash \mathsf{Ord} : \square} \qquad \frac{\Theta \vdash N : \mathsf{Ord}}{\Theta \vdash (< N) : \square}$$

$$\frac{}{(\vdash)\,\mathbf{seq}} \qquad \frac{\Theta \vdash A : \star \quad (\Gamma \vdash_\Theta \Delta)\,\mathbf{seq}}{(\Gamma, x : A \vdash_\Theta \Delta)\,\mathbf{seq}} \qquad \frac{\Theta \vdash A : \star \quad (\Gamma \vdash_\Theta \Delta)\,\mathbf{seq}}{(\Gamma \vdash_\Theta \alpha : A, \Delta)\,\mathbf{seq}} \qquad \frac{\Theta \vdash k : \blacksquare \quad (\vdash_\Theta)\,\mathbf{seq}}{(\vdash_{\Theta, a:k})\,\mathbf{seq}}$$

Figure 5. Kinding, sorting, and well-formed typing sequents.

$$\langle \mu\alpha.c \| E \rangle \to_{\mu_E} c\{E/\alpha\} \qquad \langle V \| \tilde{\mu}x.c \rangle \to_{\tilde{\mu}_V} c\{V/x\} \qquad \mu\alpha.\langle v \| \alpha \rangle \to_{\eta_\mu} v \qquad \tilde{\mu}x.\langle x \| e \rangle \to_{\eta_{\tilde{\mu}}} e$$

$$\langle \mathsf{K}^{\vec{B}}(\vec{E}, \vec{V}) \| \tilde{\mu}[\mathsf{K}^{\overrightarrow{b:k}}(\vec{\alpha}, \vec{x}).c | \ldots] \rangle \to_{\beta^{\mathsf{K}}_{\mathcal{S}}} c\{\overrightarrow{B/b}, \overrightarrow{E/\alpha}, \overrightarrow{V/x}\} \qquad \langle \mu(\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}, \vec{\alpha}].c | \ldots) \| \mathsf{H}^{\vec{B}}[\vec{V}, \vec{E}] \rangle \to_{\beta^{\mathsf{H}}_{\mathcal{S}}} c\{\overrightarrow{B/b}, \overrightarrow{V/x}, \overrightarrow{E/\alpha}\}$$

$$C^{\mathsf{K}}_{\varsigma} ::= \mathsf{K}^{\vec{B}}(\vec{E}, \square, \vec{e}, \vec{v}) \mid \mathsf{K}^{\vec{B}}(\vec{E}, \vec{V}, \square, \vec{v}) \qquad C^{\mathsf{H}}_{\varsigma} ::= \mathsf{H}^{\vec{B}}[\vec{V}, \square, \vec{v}, \vec{e}] \mid \mathsf{H}^{\vec{B}}[\vec{V}, \vec{E}, \square, \vec{e}]$$

$$C^{\mathsf{K}}_{\varsigma}[v] \to_{\varsigma^{\mathsf{K}}_{\mathcal{S}}} \mu\alpha.\langle v \| \tilde{\mu}y.\langle C^{\mathsf{K}}_{\varsigma}[y] \| \alpha \rangle \rangle \qquad C^{\mathsf{H}}_{\varsigma}[v] \to_{\varsigma^{\mathsf{H}}_{\mathcal{S}}} \tilde{\mu}x.\langle v \| \tilde{\mu}y.\langle x \| C^{\mathsf{H}}_{\varsigma}[y] \rangle \rangle \qquad \mathbf{where}\, v \notin Value$$

$$C^{\mathsf{K}}_{\varsigma}[e] \to_{\varsigma^{\mathsf{K}}_{\mathcal{S}}} \mu\alpha.\langle \mu\beta.\langle C^{\mathsf{K}}_{\varsigma}[\beta] \| \alpha \rangle \| e \rangle \qquad C^{\mathsf{H}}_{\varsigma}[e] \to_{\varsigma^{\mathsf{H}}_{\mathcal{S}}} \tilde{\mu}x.\langle \mu\beta.\langle x \| C^{\mathsf{H}}_{\varsigma}[\beta] \rangle \| e \rangle \qquad \mathbf{where}\, e \notin CoValue$$

Figure 6. Parametric rewriting theory for $\mu\tilde{\mu}_{\mathcal{S}}$.

$$\mu(\mathsf{Rise}^{j<N}[\alpha](x).c) \to \mu(\mathsf{Rise}^{i<N}[\alpha].c\{i/j, \mu(\mathsf{Rise}^{j<i}[\alpha](x).c)/x\}) \qquad \tilde{\mu}[\mathsf{Fall}^{j<N}(x)[\alpha].c] \to \tilde{\mu}[\mathsf{Fall}^{i<N}(x).c\{i/j, \tilde{\mu}[\mathsf{Fall}^{j<i}(x)[\alpha].c]\}]$$

$$\langle V \| \mathsf{Up}^0[E] \rangle \to c_0\{E/\alpha\} \qquad \langle V \| \mathsf{Up}^{M+1}[E] \rangle \to \langle \mu\beta.\langle V \| \mathsf{Up}^M[\beta] \rangle \| \tilde{\mu}x.c_1\{M/j, E/\alpha\} \rangle \qquad \mathbf{where}\, V = \mu(\mathsf{Up}^0[\alpha].c_0 | \mathsf{Up}^{j+1}[\alpha](x).c_1)$$

$$\langle \mathsf{Down}^0(V) \| E \rangle \to c_0\{V/x\} \quad \langle \mathsf{Down}^{M+1}(V) \| E \rangle \to \langle \mu\alpha.c_1\{M/j, V/x\} \| \tilde{\mu}y.\langle \mathsf{Down}^M(y) \| E \rangle \rangle \quad \mathbf{where}\, E = \tilde{\mu}[\mathsf{Down}^0.c_0 | \mathsf{Down}^{j+1}(x)[\alpha].c_1]$$

Figure 7. Rewriting theory for recursion in $\mu\tilde{\mu}_{\mathcal{S}}$.

$$V \in Value_{\mathcal{V}} ::= x \mid \mathsf{K}(\vec{e}, \vec{V}) \mid \mu(\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}, \vec{\alpha}].c | \ldots) \qquad V \in Value_{\mathcal{N}} ::= v$$

$$E \in CoValue_{\mathcal{V}} ::= e \qquad E \in CoValue_{\mathcal{N}} ::= \alpha \mid \tilde{\mu}[\mathsf{K}^{\overrightarrow{b:k}}(\vec{\alpha}, \vec{x}).c | \ldots] \mid \mathsf{H}[\vec{v}, \vec{E}]$$

Figure 8. The call-by-value ($\mathcal{V}$) and call-by-name ($\mathcal{N}$) strategies.

$C^{\mathsf{K}}_{\varsigma}$ and $C^{\mathsf{H}}_{\varsigma}$ for each (co-)constructor, and their role is to bring work to the top of a command, so that it can take over.

To implement recursion in the rewriting theory, we use the additional rules shown in Figure 7. The recursive case abstractions for Ascend and $\mathsf{Descend}$ are simplified by "unrolling" their loop: 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 around regardless of the value of the index. Contrarily, the $\mathsf{Inflate}$ and $\mathsf{Deflate}$ 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 $\mathsf{Up}^{M+1}[E]$ in tandem, unlike noetherian recursion where the two steps can be performed independently.

We also have a restriction on reduction, following the motto "don't touch unreachable branches," to ensure strong normalization. Reduction may normally occur in all contexts, except for reduction inside a case abstraction which requires an additional *reachability* caveat about the kinds of quantified types introduced by pattern matching. This restriction prevents unnecessary infinite unrolling that would otherwise occur in simple commands like $\langle length \| \mathsf{Rise}^i[\alpha] \rangle$. 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 reductions following the reachability caveat are defined as:

$$\frac{c \to c' \quad b : \overrightarrow{k \to}(< N) \in \Theta \implies N = \infty \vee N = M+1}{\mu(\mathsf{H}^\Theta[\vec{x}, \vec{\alpha}].c | \ldots) \to \mu(\mathsf{H}^\Theta[\vec{x}, \vec{\alpha}].c' | \ldots)}$$

$$\frac{c \to c' \quad b : \overrightarrow{k \to}(< N) \in \Theta \implies N = \infty \vee N = M+1}{\tilde{\mu}[\mathsf{K}^\Theta(\vec{\alpha}, \vec{x}).c | \ldots] \to \tilde{\mu}[\mathsf{K}^\Theta(\vec{\alpha}, \vec{x}).c' | \ldots]}$$

We also define the type erasure operation on programs, $Erase(c)$, 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 $\mu\tilde{\mu}_{\mathcal{S}}$-calculus is the same, except that the reachability caveat is enhanced to never reduce inside case abstractions. This means that every step of a type-erased command is justified by the same step in the original command, so that type-erasure cannot introduce infinite loops.

To demonstrate strong normalization, we use a combination of techniques. Giving a semantics for types based on Barbanera and

Berardi's symmetric candidates [4], a variant of Girard's reducibility candidates [9], as well as Krivine's classical realizability [13], an application of bi-orthogonality, establishes strong normalization of well-typed commands. Of note, the strong normalization of well-typed commands is parameterized by a strategy, which is enabled by the parameterization of the rewriting theory. Thus, instead of showing strong normalization of these related rewriting theories one-by-one, we establish strong normalization in one fell swoop by characterizing the properties of a strategy that are important for strong normalization. First, the chosen strategy $\mathcal{S}$ must be *stable*, meaning that (co-)values are closed under reduction and substitution, and non-(co-)values are closed under substitution and ς reduction. Second, $\mathcal{S}$ must be *focalizing*, meaning that (co-)variables, structures built from other (co-)values, and case abstractions must all be (co-)values. The latter criteria comes from focalization in logic [7, 16, 21]—each criterion comes from an inference rule for typing a (co-)value in focus.

Theorem 1. *For any stable and focalizing strategy $\mathcal{S}$, if $c : \Gamma \vdash_\Theta \Delta$ and $(\Gamma \vdash_\Theta \Delta)\,\mathbf{seq}$, then c is strongly normalizing in the $\mu\tilde{\mu}_\mathcal{S}$-calculus. Furthermore, $Erase(c)$ is strongly normalizing in the type-erased $\mu\tilde{\mu}_\mathcal{S}$-calculus.*

Note that the call-by-name, call-by-value, and call-by-need strategies from [8] are all stable and focalizing, so that as a corollary, we achieve strong normalization for these particular instances of the parametric $\mu\tilde{\mu}_\mathcal{S}$-calculus. Furthermore, the "maximally" non-deterministic strategy—attained by letting every term be a value and every co-term be a co-value—is also stable and focalizing, which gives another account of strong normalization for the symmetric λ-calculus [14] as a corollary.

5.1 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 2. We will see how the intuitive argument for termination can be represented using the type indices for recursion in various ways.

Example 4. Recall the $length$ function from Example 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 $\mu\tilde{\mu}_\mathcal{S}$-calculus, we can give a more precise and non-recursive definition for $length$:

$$
\begin{aligned}
&length : \forall a : \star.\, \mathsf{Ascend}\, i < \infty.\, \mathsf{List}(i,a) \to \mathsf{Nat}(i)\\
&length = \mu(\mathsf{Spec}^a[\mathsf{Rise}^{i<\infty}[\mathsf{Nil}\cdot\gamma](r)].\langle \mathsf{Z} \| \gamma\rangle\\
&\qquad|\mathsf{Spec}^a[\mathsf{Rise}^{i<\infty}[\mathsf{Cons}^{j<i}(x,xs)\cdot\gamma](r)].\\
&\qquad\quad\langle r \| \mathsf{Rise}^j[xs\cdot\tilde{\mu}y.\langle \mathsf{S}^j(y)\|\gamma\rangle]\rangle)
\end{aligned}
$$

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 Ascend co-data type. Going further, we may unravel the deep patterns into shallow case analysis, giving annotations on the introduction of every co-variable:

$$
\begin{aligned}
&length = \mu(\mathsf{Spec}^a[\alpha^{\mathsf{Ascend}\,i<\infty.\,\mathsf{List}(i,a)\to\mathsf{Nat}(i)}].\\
&\langle\mu(\mathsf{Rise}^{i<\infty}[\beta^{\mathsf{List}(i,a)\to\mathsf{Nat}(i)}](r^{\mathsf{Ascend}\,j<i.\,\mathsf{List}(j,a)\to\mathsf{Nat}(j)}).\\
&\quad\langle\mu([xs^{\mathsf{List}(i,a)}\cdot\gamma^{\mathsf{Nat}(i)}].\,\langle xs|\\
&\qquad|\tilde{\mu}[\mathsf{Nil}.\langle\mathsf{Z}\|\gamma\rangle\\
&\qquad\quad|\mathsf{Cons}^{j<i}(x^a_{,} ys^{\mathsf{List}(j,a)}).\langle r\|\mathsf{Rise}^j[ys\cdot\tilde{\mu}y^{\mathsf{Nat}(j)}.\langle\mathsf{S}^j(y)\|\gamma\rangle]\rangle]\rangle)|\\
&\quad|\beta\rangle)|\\
&|\alpha\rangle)
\end{aligned}
$$

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* 4.

Example 5. Recall the $countUp$ function from Example 2. When we attempt to encode this function into the $\mu\tilde{\mu}_\mathcal{S}$-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 $\mathsf{ANat} = \mathsf{Descend}\, i < \infty.\, \mathsf{Nat}(i)$. We can then describe $countUp$ as a function from ANat to a $\mathsf{Stream}(i, \mathsf{ANat})$ by noetherian recursion on i:

$$
\begin{aligned}
&countUp : \mathsf{Ascend}\, i<\infty.\,\mathsf{ANat}\to\mathsf{Stream}(i,\mathsf{ANat})\\
&countUp = \mu(\mathsf{Rise}^{i<\infty}[x\cdot\mathsf{Head}[\alpha]](r).\langle x\|\alpha\rangle\\
&\qquad|\mathsf{Rise}^{i<\infty}[\mathsf{Fall}^{j<i}(x)\cdot\mathsf{Tail}^{k<i}[\beta]](r).\\
&\qquad\quad\langle r\|\mathsf{Rise}^k[\mathsf{Fall}^{j+1}(\mathsf{S}^j(x))\cdot\beta]\rangle)
\end{aligned}
$$

End example 5.

Example 6. The previous example shows how infinite streams may be modeled by co-data. However, recall the other approach to infinite objects mentioned in Example 3. 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 $\mathsf{InfList}(a) = \mathsf{Inflate}\, i : \mathsf{Ix}\,.\,\mathsf{IxList}(i,a)$, we can encode $zeroes$ as:

$$
\begin{aligned}
&zeroes : \mathsf{InfList}(\mathsf{Nat}(0))\\
&zeroes = \mu(\mathsf{Up}^0[\alpha^{\mathsf{IxList}(0,\mathsf{Nat})}].\langle\mathsf{Nil}\|\alpha\rangle\\
&\qquad|\mathsf{Up}^{i+1}[\alpha^{\mathsf{IxList}(i+1,\mathsf{Nat})}](r^{\mathsf{IxList}(i,\mathsf{Nat})}).\langle\mathsf{Cons}(\mathsf{Z},r)\|\alpha\rangle)
\end{aligned}
$$

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:

$$
\begin{aligned}
&cat : \forall a:\star.\,\mathsf{InfList}(a)\to\mathsf{InfList}(a)\to\mathsf{InfList}(a)\\
&cat = \langle\mu(\mathsf{Spec}^a[xs\cdot ys\cdot\mathsf{Up}^i[\alpha]].\\
&\qquad\langle xs\|\mathsf{Up}^i[\tilde{\mu}zs.\langle cat'\|\mathsf{Up}^i[zs\cdot ys\cdot\alpha]\rangle]\rangle)
\end{aligned}
$$

$$
\begin{aligned}
&cat' : \forall a:\star.\,\mathsf{Inflate}\,i:\mathsf{Ix}\,.\,\mathsf{IxList}(i,a)\to\mathsf{InfList}(a)\to\mathsf{IxList}(i,a)\\
&cat' = \mu(\mathsf{Spec}^a[\mathsf{Up}^0[\mathsf{Nil}\cdot ys\cdot\alpha]].\langle\mathsf{Nil}\|\alpha\rangle\\
&\qquad|\mathsf{Spec}^a[\mathsf{Up}^{i+1}[\mathsf{Nil}\cdot ys\cdot\alpha](r)].\langle ys\|\mathsf{Up}^{i+1}[\alpha]\rangle\\
&\qquad|\mathsf{Spec}^a[\mathsf{Up}^{i+1}[\mathsf{Cons}(x,xs)\cdot ys\cdot\alpha](r)].\\
&\qquad\quad\langle\mathsf{Cons}(x,\mu\beta.\langle r\|xs\cdot ys\cdot\beta\rangle)\|\alpha\rangle)
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathbf{codata}\,\mathsf{StopStream}(i<\infty, a:\star)\,\mathbf{where}\\
&\quad\mathsf{Head}: \ |\,\mathsf{StopStream}(i,a)\vdash a\\
&\quad\mathsf{Tail}: \ |\,\mathsf{StopStream}(i,a)\vdash_{j<i} 1,\mathsf{StopStream}(j,a)
\end{aligned}
$$

A $\mathsf{StopStream}(i,a)$ object is like a $\mathsf{Stream}(i,a)$ 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:

$$\langle cat \| xs \cdot ys \cdot \mathsf{Head}[\alpha] \rangle = \langle xs \| \mathsf{Head}[\alpha] \rangle$$
$$\langle cat \| xs \cdot ys \cdot \mathsf{Tail}[\delta, \beta] \rangle = \langle cat \| \mu\gamma. \langle xs \| \mathsf{Tail}[\tilde{\mu}[().\langle ys \| \beta \rangle], \gamma] \rangle \cdot ys \cdot \beta \rangle$$

This function encodes into a similar pair of worker-wrapper values, where now a possibly infinite list is represented as a terminating stream $\mathsf{InfList}(a) = \mathsf{Ascend}\, i < \infty.\, \mathsf{StopStream}(i, a)$:

$$\begin{aligned}
cat' &: \forall a : \star.\, \mathsf{Ascend}\, i < \infty. \\
&\qquad \mathsf{StopStream}(i, a) \to \mathsf{InfList}(a) \to \mathsf{StopStream}(i, a) \\
cat' &= \mu(\mathsf{Spec}^a[\mathsf{Rise}^{i<\infty}[xs \cdot ys \cdot \mathsf{Head}[\alpha]](r)].\langle xs \| \alpha \rangle \\
&\qquad |\mathsf{Spec}^a[\mathsf{Rise}^{i<\infty}[xs \cdot ys \cdot \mathsf{Tail}^{j<i}[\delta, \beta]](r)]. \\
&\qquad\quad \langle r \| \mathsf{Rise}^j[\mu\gamma.\langle xs \| \mathsf{Tail}^j[\tilde{\mu}[().\langle ys \| \mathsf{Rise}^j[\beta] \rangle], \gamma] \rangle \cdot ys \cdot \beta] \rangle)
\end{aligned}$$

End example 6.

Intermezzo 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 [18]. 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-fixpoint 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 intermezzo* 1.

Example 7. 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:

$$\begin{aligned}
ack &: \mathsf{Ascend}\, i < \infty.\, \mathsf{Ascend}\, j < \infty.\, \mathsf{Nat}(i) \to \mathsf{Nat}(j) \to \mathsf{ANat} \\
ack &= \mu(\mathsf{Rise}^{i<\infty}[\mathsf{Rise}^{j<\infty}[\mathsf{Z} \cdot y \cdot \alpha](r_2)](r_1).\langle \mathsf{Fall}^{j+1}(\mathsf{S}^j(y)) \| \alpha \rangle \\
&\qquad |\mathsf{Rise}^{i<\infty}[\mathsf{Rise}^{j<\infty}[\mathsf{S}^{i'<i}(x) \cdot \mathsf{Z} \cdot \alpha](r_2)](r_1). \\
&\qquad\quad \langle r_1 \| \mathsf{Rise}^{i'}[\mathsf{Rise}^1[x \cdot \mathsf{S}^0(\mathsf{Z}) \cdot \alpha]] \rangle \\
&\qquad |\mathsf{Rise}^{i<\infty}[\mathsf{Rise}^{j<\infty}[\mathsf{S}^{i'<i}(x) \cdot \mathsf{S}^{j'<j}(y) \cdot \alpha](r_2)](r_1). \\
&\qquad\quad \langle r_2 \| \mathsf{Rise}^{j'}[\mathsf{S}^{i'}(x) \cdot y \cdot \tilde{\mu}[\mathsf{Fall}^{k<\infty}(z). \\
&\qquad\qquad\qquad \langle r_1 \| \mathsf{Rise}^{i'}[\mathsf{Rise}^k[x \cdot z \cdot \alpha]] \rangle]]] \rangle)
\end{aligned}$$

Essentially, we get two recursive arguments from nesting Ascend:

$$r_1 : \mathsf{Ascend}\, i' < i.\, \mathsf{Ascend}\, j < \infty.\, \mathsf{Nat}(i') \to \mathsf{Nat}(j) \to \mathsf{ANat}$$
$$r_2 : \mathsf{Ascend}\, j' < j.\, \mathsf{Nat}(i) \to \mathsf{Nat}(j') \to \mathsf{ANat}$$

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, $\mathsf{Descend}$, 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 $\mathsf{Descend}$-ed 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* 7.

6. Natural Deduction and Effect-Free Programs

So far, we have looked at a calculus for representing recursion via structures in sequent style, which corresponds to a classical logic and thus includes control effects [11]. Let's now briefly shift focus, and see how the intuition we gained from the sequent calculus can be reflected back into a more traditional core calculus for expressing functional-style recursion. The goal here is to see how the recursive 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 $\mu\tilde{\mu}_S$-calculus corresponding to *minimal* logic.

Essentially, the minimal restriction of the $\mu\tilde{\mu}_S$-calculus for representing effect-free functional programs follows a single mantra, based on the connection between classical and minimal logics: there is always *exactly* one conclusion. In the type system, this means that the sequent for typing terms has the more restricted form $\Gamma \vdash_\Theta v : A$, 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 type constructors we can express. For example, common sums and products, declared as

$$\begin{array}{llll}
\mathbf{data}\, a \oplus b\, \mathbf{where} & & \mathbf{codata}\, a \,\&\, b\, \mathbf{where} & \\
\quad \mathsf{Left} : & a \vdash a \oplus b | & \quad \mathsf{Fst} : & |a \,\&\, b \vdash a \\
\quad \mathsf{Right} : & b \vdash a \oplus b | & \quad \mathsf{Snd} : & |a \,\&\, b \vdash b
\end{array}$$

fit into this restricted typing discipline, because each of their (co-)constructors only ever involves one type to the right of entailment. However, the (co-)data types for representing more exotic connectives like subtraction and linear logic's par

$$\begin{array}{llll}
\mathbf{data}\, a - b\, \mathbf{where} & & \mathbf{codata}\, a \parr b\, \mathbf{where} & \\
\quad \mathsf{Pause} : & a \vdash a - b | b & \quad \mathsf{Split} : & |a \parr b \vdash a, b
\end{array}$$

do not fit, because they require placing two types to the right of entailment. In sequent style, this means these *minimal* data types can never contain a co-value, and *minimal* 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 $a \,\&\, b$, we could access the first component as a record field, $v.\mathsf{Fst}$, and we describe an object of this type by saying how it responds to all possible observations, $\{\mathsf{Fst} \Rightarrow v_1 | \mathsf{Snd} \Rightarrow v_2\}$, with the typing rules:

$$\frac{\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : B}{\Gamma \vdash \{\mathsf{Fst} \Rightarrow v_1 | \mathsf{Snd} \Rightarrow v_2\} : A \,\&\, B} \quad \frac{\Gamma \vdash v : A \,\&\, B}{\Gamma \vdash v.\mathsf{Fst} : A} \quad \frac{\Gamma \vdash v : A \,\&\, B}{\Gamma \vdash v.\mathsf{Snd} : B}$$

Likewise, the traditional λ-abstractions and type abstractions from System F can be expressed by objects of these form. Specifically, since they are user-definable, minimal co-data types with one constructor, $\mathsf{Call} : a|a \to b \vdash b$ and $\mathsf{Spec} : |\forall a \vdash_{b:\star} a\ b$, the abstractions can be given as syntactic sugar:

$$\lambda x^A.v = \{\mathsf{Call}[x^A] \Rightarrow v\} \qquad \Lambda b^\star.v = \{\mathsf{Spec}^{b:\star} \Rightarrow v\}$$

Thus, these objects also serve as "generalized λ-abstractions" [2] defined by shallow case analysis rather than deep pattern-matching.

The typing rules for recursive structures translated to functional style are shown in Figure 9, and the reduction rules for the calculus

$$\frac{\Gamma \vdash_\Theta v_0 : A\{0/i\} \quad \Gamma, x : A\{j/i\} \vdash_{\Theta, j:\mathsf{Ix}} v_1 : A\{j+1/i\}}{\Gamma \vdash_\Theta \{\mathsf{Up}^0 \Rightarrow v_0 |\, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\} : \mathsf{Inflate}\, i : \mathsf{Ix}\,.A} \qquad \frac{\Gamma \vdash_\Theta v : \mathsf{Inflate}\, i : \mathsf{Ix}\,.A \quad \Theta \vdash M : \mathsf{Ix}}{\Gamma \vdash_\Theta v.\,\mathsf{Up}^M : A\{M/i\}}$$

$$\frac{\Gamma \vdash_\Theta v : A\{M/i\} \quad \Theta \vdash M : \mathsf{Ix}}{\Gamma \vdash_\Theta \mathsf{Down}^M(v) : \mathsf{Deflate}\, i : \mathsf{Ix}\,.A} \qquad \frac{\Gamma \vdash_\Theta v : \mathsf{Deflate}\, i : \mathsf{Ix}\,.A \quad \Gamma, x : A\{0/i\} \vdash_\Theta v_0 : C \quad \Gamma, x : A\{j+1/i\} \vdash_{\Theta, j:\mathsf{Ix}} v_1 : A\{j/i\}}{\Gamma \vdash_\Theta \mathbf{loop}\, v\, \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 |\, \mathsf{Down}^{j+1}(x) \Rightarrow v_1 : C}$$

$$\frac{\Gamma, x : \mathsf{Ascend}\, i < j.A \vdash_{\Theta, j<N} v : A\{j/i\}}{\Gamma \vdash_\Theta \{\mathsf{Rise}^{j<N}(x) \Rightarrow v\} : \mathsf{Ascend}\, i < N.A} \qquad \frac{\Gamma \vdash_{\Theta, j<N} v : A\{j/i\}}{\Gamma \vdash_\Theta \{\mathsf{Rise}^{j<N} \Rightarrow v\} : \mathsf{Ascend}\, i < N.A} \qquad \frac{\Gamma \vdash_\Theta v : \mathsf{Ascend}\, i < N.A \quad \Theta \vdash M < N}{\Gamma \vdash_\Theta v.\,\mathsf{Rise}^M : A\{M/i\}}$$

Figure 9. Typing primitive and noetherian recursion in natural deduction style.

$$\{\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}] \Rightarrow v' | \ldots\}.\mathsf{H}^{\vec{B}}[\vec{v}] \to v'\{\overrightarrow{B/b}, \overrightarrow{v/x}\} \qquad \mathbf{case}\ \mathsf{K}^{\vec{B}}(\vec{v})\, \mathbf{of}\ \mathsf{K}^{\overrightarrow{b:k}}(\vec{x}) \Rightarrow v' | \ldots \to v'\{\overrightarrow{B/b}, \overrightarrow{v/x}\}$$

$$\{\mathsf{Rise}^{j<N}(x) \Rightarrow v\} \to \{\mathsf{Rise}^{i<N} \Rightarrow v\{i/j, \{\mathsf{Rise}^{j<i}(x) \Rightarrow v\}/x\}\}$$

$$\{\mathsf{Up}^0 \Rightarrow v_0 |\, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\mathsf{Up}^0 \to v_0 \qquad \{\mathsf{Up}^0 \Rightarrow v_0 |\, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\mathsf{Up}^{M+1} \to v_1\{M/j, \{\mathsf{Up}^0 \Rightarrow v_0 |\, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\mathsf{Up}^M/x\}$$

$$\mathbf{loop}\ \mathsf{Down}^0(v)\, \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 |\, \mathsf{Down}^{j+1}(x) \Rightarrow v_1 \to v_0\{v/x\}$$

$$\mathbf{loop}\ \mathsf{Down}^{M+1}(v)\, \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 |\, \mathsf{Down}^{j+1}(x) \Rightarrow v_1 \to \mathbf{loop}\ \mathsf{Down}^M(v_1\{M/j, v/x\})\, \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 |\, \mathsf{Down}^{j+1}(x) \Rightarrow v_1$$

Figure 10. Reduction rules for a natural deduction language with (co-)data types and recursion.

$$x^\flat = x \qquad (\mathsf{K}^{\vec{B}}(\vec{v}))^\flat = \mathsf{K}^{\vec{B}}(\overrightarrow{v^\flat}) \qquad (\mathbf{case}\, v\, \mathbf{of}\ \mathsf{K}^{\overrightarrow{b:k}}(\vec{x}) \Rightarrow v' | \ldots)^\flat = \mu\alpha.\langle v^\flat \| \tilde{\mu}[\mathsf{K}^{\overrightarrow{b:k}}(\vec{x}).\langle v'^\flat \| \alpha\rangle | \ldots]\rangle$$

$$(v'.\mathsf{H}^{\vec{B}}[\vec{v}])^\flat = \mu\alpha.\langle v'^\flat \| \mathsf{H}^{\vec{B}}[\overrightarrow{v^\flat}, \alpha]\rangle \qquad \{\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}] \Rightarrow v | \ldots\}^\flat = \mu(\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}, \alpha].\langle v^\flat \| \alpha\rangle | \ldots) \qquad \{\mathsf{Rise}^{j<N}(x) \Rightarrow v\}^\flat = \mu(\mathsf{Rise}^{j<N}[\alpha](x).\langle v^\flat \| \alpha\rangle)$$

$$\{\mathsf{Up}^0 \Rightarrow v_0 |\, \mathsf{Up}^{j+1}(x) \Rightarrow v_1\}^\flat = \mu(\mathsf{Up}^0[\alpha].\langle v^\flat \| \alpha\rangle | \mathsf{Up}^{j+1}[\alpha](x).\langle v^\flat \| \alpha\rangle)$$

$$(\mathbf{loop}\, v\, \mathbf{of}\ \mathsf{Down}^0(x) \Rightarrow v_0 |\, \mathsf{Down}^{j+1}(x) \Rightarrow v_1)^\flat = \mu\alpha.\langle v^\flat \| \tilde{\mu}[\mathsf{Down}^0(x).\langle v_0^\flat \| \alpha\rangle | \mathsf{Down}^{j+1}(x)[\alpha].\langle v_1^\flat \| \alpha\rangle]\rangle$$

Figure 11. Type-preserving translation from a pure, natural deduction language to $\mu\tilde{\mu}_{\mathcal{S}}$.

are shown in Figure 10. Intuitively, the objects of $\mathsf{Inflate}(A)$ are stepwise loops that can return any $A\ N$ by counting up from 0 and using the previous instances of itself, while we can write looping case expressions over values of $\mathsf{Deflate}(A)$ to count down from any $A\ N$ to 0. Similarly, values of $\mathsf{Ascend}(N, A)$ are self-referential objects that always behave the same no matter the number of recursive invocations. Curiously though, the recursive forms for $\mathsf{Descend}(N, A)$ are conspicuously missing from the functional calculus. In essence, the recursive form for $\mathsf{Descend}(N, A)$ is a case expression that introduces a continuation variable for the recursive path out of the expression in addition to the normal return path, effectively requiring a form of subtraction type $C - \mathsf{Descend}(M, A)$ for smaller indices M. So while $\mathsf{Descend}$ can still be used to hide 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 or minimal settings. It occurs with De Morgan laws ($\neg(A \wedge B) \to (\neg A) \vee (\neg B)$ is not intuitionistically valid), the conjunctive and disjunctive connectives of linear logic ($⅋$ requires multiple conclusions so it does not fit the minimal mold), and here as well.

Intuitively, we can think of the values of $\mathsf{Inflate}(A)$ as a dependently typed version of the recursion operator for natural numbers in Gödel's System T [10]. Indeed, we can encode such an operator:

$$\begin{aligned} &rec : \forall a : \mathsf{Ix} \to \star. \\ &\qquad a\ 0 \to (\mathsf{Inflate}\, i : \mathsf{Ix}\,.a\ i \to a\ (i+1)) \to \mathsf{Inflate}\, i : \mathsf{Ix}\,.a\ i \\ &rec = \lambda a\ x\ f.\{\mathsf{Up}^{0:\mathsf{Ix}} \Rightarrow x |\, \mathsf{Up}^{j+1:\mathsf{Ix}}(r) \Rightarrow f.\,\mathsf{Up}^j\ r\} \end{aligned}$$

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 $rec\ a\ x\ f \twoheadrightarrow rec_{a,x,f}$ we have

$$rec_{a,x,f}.\mathsf{Up}^0 \to x \qquad rec_{a,x,f}.\mathsf{Up}^{M+1} \to f.\mathsf{Up}^M\ (rec_{a,x,f}.\mathsf{Up}^M)$$

Contraposed, $\mathsf{Deflate}(A)$ implements a dependently-typed, stepwise recursion going the other way. The looping form breaks down a value depending on an arbitrary index N 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, $v : \mathsf{Vec}(N, \mathsf{ANat})$, in accumulator style by looping over the recursive structure $\mathsf{Descend}\, i : \mathsf{Ix}\,.\,\mathsf{ANat} \otimes \mathsf{Vec}(i, \mathsf{ANat})$:[5]

$$\begin{aligned} &\mathbf{loop}\ \mathsf{Down}^N(\mathsf{Fall}^0(\mathsf{Z}), v)\ \mathbf{of} \\ &\qquad \mathsf{Down}^0(acc, \mathsf{Nil}) \Rightarrow acc \\ &\qquad |\, \mathsf{Down}^{i+1}(acc, \mathsf{Cons}(x, xs)) \Rightarrow (x + acc, xs) \end{aligned}$$

Instead, values of Ascend are useful for representing stronger induction that recurses on deeply nested sub-structures. For example, we can convert a list $x_1, x_2, \ldots, x_n$ into a list of its adjacency pairs $(x_1, x_2), (x_3, x_4), \ldots, (x_{n-1}, x_n)$ by

$$\begin{aligned} &pairs\ \mathsf{Nil} &&= \mathsf{Nil} \\ &pairs\ \mathsf{Cons}(x, ys) &&= \mathsf{Nil} \\ &pairs\ \mathsf{Cons}(x, \mathsf{Cons}(y, zs)) &&= \mathsf{Cons}((x, y), pairs\ zs) \end{aligned}$$

[5] Note, we assume an addition operator $+ : \mathsf{ANat} \to \mathsf{ANat} \to \mathsf{ANat}$.

where we silently drop the final element if the list is odd. The $pairs$ function can be straightforwardly encoded using Ascend as:

$$pairs : \forall a : \star.\, \mathsf{Ascend}\, i < \infty.\, \mathsf{List}(i, a) \to \mathsf{List}(i, a \otimes a)$$

$$\begin{aligned} pairs = \lambda a^{\star}.\{&\mathsf{Rise}^{i<\infty}(r) \Rightarrow \lambda x^{\mathsf{List}(i,a)}.\, \mathbf{case}\, xs\, \mathbf{of} \\ &\quad \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ &\quad \mathsf{Cons}^{j<i}(x^a, ys^{\mathsf{List}(j,a)}) \Rightarrow \mathbf{case}\, ys\, \mathbf{of} \\ &\qquad \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ &\qquad \mathsf{Cons}^{k<j}(y^a, zs^{\mathsf{List}(k,a)}) \Rightarrow \mathsf{Cons}^k((x,y), r.\mathsf{Rise}^k zs)\} \end{aligned}$$

Note that the type of the recursive argument r is $\mathsf{Ascend}\, i' < i.\, \mathsf{List}(i', a) \to \mathsf{List}(i', a \otimes a)$. Thus, the recursive self-invocation $r.\mathsf{Rise}^k : \mathsf{List}(k, a) \to \mathsf{List}(k, a \otimes a)$ is well-typed, since we learn that $j < i$ and $k < j$ by analyzing the Cons structure of the list and learn that $k < i$ by transitivity.

Finally, note that we can translate this functional calculus into the minimal subset of the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus, as shown in Figure 11. This translation is type-preserving, and each of the source reductions maps to at least one reduction in the call-by-name instance of $\mu\tilde{\mu}_{\mathcal{S}}$ [8], $\mu\tilde{\mu}_{\mathcal{N}}$, where the set of values is as large as possible and includes every term. So, because the $\mu\tilde{\mu}_{\mathcal{N}}$-calculus does not allow for well-typed infinite loops, neither does its functional counterpart.

Theorem 2. *If* $\Gamma \vdash_{\Theta} v : A$ *and* $(\Gamma \vdash_{\Theta} \alpha : A)\, \mathbf{seq}$ *are derivable then* v *is strongly normalizing.*

7. Conclusion

Co-induction need not be a second-class citizen compared to induction in programming languages. Dedication to duality provides the key for unlocking co-recursion from recursion as its equal and opposite force. We are able to freely mix inductive and co-inductive styles of programming along with computational effects (specifically, classical control effects) without losing properties like strong normalization or extensional reasoning. Additionally, we show how the lessons we learn can be translated back to the more familiar ground of effect-free functional programming, although its inherent lack of duality causes some symmetries of recursion schemes to be lost in translation. We can write pure functional programs with mixed induction and co-induction, but the asymmetry of the paradigm blocks the full expression of certain recursion principles.

In order to ensure that recursion is well-founded, we use type-level indices indicating the size of types as a tool. This is a pragmatic choice: the nature of computation in the sequent calculus makes it essential to track size arguments for well-foundedness “inside” larger structures. Allowing size information to flow into structures is a natural consequence of the co-data presentation of functions. Implementations of type theory typically check the arguments to a recursive function definition, but since functions are just another user-defined co-data structure containing these arguments, there is no inherent reason to limit this functionality to function types alone.

We have shown how both recursion and co-recursion in programs can be drawn from the mathematical principles of primitive and noetherian induction, and codified as programming structures for representing recursive processes. The style of primitive recursion with computationally sensitive type-level indices can be mixed with noetherian recursion that use computationally-irrelevant indices. We see that the primitive and noetherian recursion principles, which are generally distinct mathematically, are also distinct computationally and have different uses. The general (co-)data mechanism helped us to understand these principles for recursion in programs, but the recursors were generated by hand. Can we find the general mechanism that encompasses recursion in programs, in the same way that we have encompassed recursion in (co-)data types?

A clear subject for future study is to enrich the existing dependencies in types to be closer to full-spectrum dependent types. We find that a modest amount of dependency in primitive recursion, in the form of numeric type indices admitting case analysis, helps us encode programs over Haskell-style infinite lists. Further exploring the nature of this dependency may show how to adapt this theory to be applicable to the use in proof assistants with dependent types. We also saw how the duality of classical logic is useful in the study of recursion. Can this classicality be rectified with more complex notions of dependency, so that dependent types can be given a computational view of classical reasoning principles?

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback on improving this paper. Paul Downen and Zena M. Ariola have been supported by NSF grant CCF-1423617.

References

[1] A. Abel. *A Polymorphic Lambda Calculus with Sized Higher-Order Types*. Ph.D. thesis, Ludwig-Maximilians-Universität München, 2006.

[2] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In *ICFP*, 2013.

[3] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *POPL*, 2013.

[4] F. Barbanera and S. Berardi. A symmetric lambda calculus for "classical" program extraction. In *TACS ’94*, pages 495–515, 1994.

[5] T. Coquand and P. Dybjer. Inductive definitions and type theory an introduction. In *FSTTCS*, volume 880 of *LNCS*, 1994.

[6] P.-L. Curien and H. Herbelin. The duality of computation. In *International Conference on Functional Programming*, pages 233–243, 2000.

[7] P.-L. Curien and G. Munch-Maccagnoni. The duality of computation under focus. *Theoretical Computer Science*, pages 165–181, 2010.

[8] P. Downen and Z. M. Ariola. The duality of construction. In *European Symposium on Programming*, 2014.

[9] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, 1989.

[10] K. Gödel. On a hitherto unexploited extension of the finitary standpoint. *Journal of Philosophical Logic*, 9(2):133–142, 1980.

[11] T. Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58, 1990.

[12] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, 1987.

[13] J.-L. Krivine. Realizability in classical logic. In *Interactive models of computation and program behaviour*, volume 27, pages 197–229. Société Mathématique de France, 2009.

[14] S. Lengrand and A. Miquel. Classical Fω, orthogonality and symmetric candidates. *Annals of Pure and Applied Logic*, 153(1):3–20, 2008.

[15] P. Martin-Löf. A theory of types. Technical Report 71-3, University of Stockholm, 1971.

[16] G. Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, pages 409–423. Springer, 2009.

[17] P. M. Nax. *Inductive Definition in Type Theory*. Ph.D. thesis, Cornell University, 1988.

[18] N. Oury. Coinductive types and type preservation. Message on the Coq-club mailing list, June 2008.

[19] S. Singh, S. P. Jones, U. Norell, F. Pottier, E. Meijer, and C. McBride. Sexy types—are we done yet? Software Summit, Apr. 2011.

[20] P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of ICFP*, pages 189–201. ACM, 2003.

[21] N. Zeilberger. On the unity of duality. *Annals of Pure Applied Logic*, 153(1-3):66–96, 2008.

A. Strong Normalization

We now give a proof sketch for strong normalization of well-typed commands of the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus (the full details of the proof follow afterward in expanded form). Our approach uses techniques based on both orthogonality [7] and a variant of Barbanera and Berardi's symmetric candidates [1]. The proof is given parametrically with respect to $\mathcal{S}$, so long as it satisfies some general conditions. First, $\mathcal{S}$ must be *stable*, meaning that (co-)values are closed under reduction and substitution, and non-(co-)values are closed under substitution and ς reduction. Second, $\mathcal{S}$ must be *focalizing*, meaning that (co-)variables, structures built from other (co-)values, and case abstractions must all be (co-)values. The latter criteria comes from focalization in logic [2, 7, 9]—each criterion comes from an inference rule for typing a (co-)value in focus. For the remainder of the section, we will assume $\mathcal{S}$ is stable and focalizing, but otherwise arbitrary.

Our main goal is now to come up with a suitable model of types that ensures strong normalization. Since types in $\mu\tilde{\mu}_{\mathcal{S}}$ classify both terms and co-terms, we build a model for types as a pair of sets of terms and co-terms, called *pre-types*. These pre-types relate to one another in two fundamental ways. On the one hand we have refinement: $\mathcal{A}$ refines $\mathcal{B}$ ($\mathcal{A} \sqsubseteq \mathcal{B}$) if all the (co-)terms of $\mathcal{A}$ are included in $\mathcal{B}$. On the other hand we have sub-typing: $\mathcal{A}$ is a subtype of $\mathcal{B}$ ($\mathcal{A} \leq \mathcal{B}$) if the terms of $\mathcal{A}$ are included in $\mathcal{B}$ and the co-terms of $\mathcal{B}$ are included in $\mathcal{A}$. That directions in sub-typing are reversed arises naturally from the view of co-terms as "experiments" or properties on values, and captures Liskov's substitution principle [6]: if $\mathcal{A} \leq \mathcal{B}$, then properties on the objects of $\mathcal{B}$ hold for $\mathcal{A}$ as well.

The fundamental operation on pre-types is orthogonality [7]: $\mathcal{A}^{\perp}$ contains all the strongly normalizing (co-)terms that form strongly normalizing commands with any (co-)term of $\mathcal{A}$. However, orthogonality alone is not enough to define the meaning of types in a classical setting [5]. For example, we need to be able to justify the Act and $CoAct$ rules for typing μ- and $\tilde{\mu}$- abstractions, which is not trivial [7], and requires us to know something more about reduction. Our main idea is that types need to be saturated under the reductions which happen at the top level (or "head") of the command. Furthermore, these head reductions can be classified into three groups based on their *charge*. In a positive head reduction, the term side of a command takes control. Dually, in a negative head reduction ($\mapsto_{-}$), the co-term side of a command takes control. More specifically, positive head reductions are given by the rules for μ_E, ς for data structures, and unfolding Ascend at the head of a command (ς head reduction occurs in commands of the form $\langle C^{\mathsf{K}}_{\varsigma}[v] \| E\rangle$ or $\langle C^{\mathsf{K}}_{\varsigma}[e] \| E\rangle$, and unfolding is similar). Dually, negative head reductions are given by the rules for $\tilde{\mu}_V$, ς for co-data structures and unfolding $\mathsf{Descend}$. Neutral head reductions ($\mapsto_0$) involve the term and co-term operating in concert, and include the remaining rules. We can then talk about the head of a pre-type $\mathcal{A}$, written $Head(\mathcal{A})$. A strongly normalizing term v is in $Head(\mathcal{A})$ whenever $\langle v \| E\rangle \mapsto_{+} c$ and c is strongly normalizing, for any co-value E in $\mathcal{A}$. The co-terms in $Head(\mathcal{A})$ are defined dually by negative head reductions.

With this machinery, we are finally ready to define the notion of reducibility candidate [4] that encompasses our intended meaning of types. A *reducibility candidate* is any pre-type $\mathcal{A}$ that is *orthogonally sound* ($\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$), *orthogonally complete* ($\mathcal{A}^{\perp} \sqsubseteq \mathcal{A}$) and includes its own head ($Head(\mathcal{A}) \sqsubseteq \mathcal{A}$). These three criteria include everything we need to know to ensure that typing implies strong normalization. Orthogonal soundness ensures us that Cut is sound. Orthogonal completeness ensures that open commands are still strongly normalizing. The inclusion of $Head$ ensures that more complex (co-)terms like the $\mu-$ and $\tilde{\mu}-$ abstractions or general (co-)data structures are included. Indeed, head reductions are used to characterize these complex (co-)terms by their behavior instead of by their syntax in a general and extensible way.

We use the fixed point construction of the symmetric candidates technique [1, 5] to simplify the process of ensuring the meaning of a type is a reducibility candidate: so long as we can fully describe a type by a pre-type of *simple* (co-)terms which can never take a positive or negative step, we can automatically generate a corresponding reducibility candidate for that type. The idea is to define the saturation of a pre-type that adds the missing (co-)terms. A strongly normalizing term v is in $Sat(\mathcal{A})$ when either $\langle v \| E\rangle$ is strongly normalizing or $\langle v \| E\rangle \mapsto_{+} c$ and c is strongly normalizing for every co-value E in $\mathcal{A}$, and dually for co-terms in $Sat(\mathcal{A})$. Sat preserves the sub-typing order of the lattice of pre-types, so we can take the fixed point (where $\sqcup$ unions the (co-)terms of pre-types).

Lemma 1. *For any $\mathcal{C}$, there is a solution to $\mathcal{A} = \mathcal{C} \sqcup Sat(\mathcal{A})$.*

Furthermore, as long as $\mathcal{C}$ is a forward closed, orthogonally sound pre-type of simple (co-)values, then this fixed point is a reducibility candidate. Thus, this fixed point construction gives us an operation, $\mathcal{R}(-)$, for generating saturated reducibility candidates from these simplified pre-types.

Types are interpreted by means of a generalized fold over the structure of their syntax. In general, we say $[\![\mathsf{F}(\overrightarrow{A})]\!]_{\mathcal{H}} = \mathcal{H}(\mathsf{F}(\overrightarrow{[\![A]\!]_{\mathcal{H}}}))$ where the map $\mathcal{H}$ determines the meaning of the constructed type from the meaning of its constituent parts. This $\mathcal{H}$ operates over *hybrid types* consisting of a *syntactic* type constructor with *semantic* sub-components. Our task is to come up with such an $\mathcal{H}$ that interprets all meaningful hybrid types. For each type constructor we give an axillary meaning function that describes the *core* meaning of hybrid types, $\langle\!\langle \mathsf{F}(\overrightarrow{\mathcal{A}})\rangle\!\rangle_{\mathcal{H}}$. Data types, like $\otimes$, are defined by their constructors:

$$v \in \langle\!\langle \mathcal{A} \otimes \mathcal{B}\rangle\!\rangle_{\mathcal{H}} \iff v \in (Cons_{\mathcal{H}}(\mathcal{A} \otimes \mathcal{B}), \emptyset)^{\perp s \perp s}$$
$$e \in \langle\!\langle \mathcal{A} \otimes \mathcal{B}\rangle\!\rangle_{\mathcal{H}} \iff e \in (Cons_{\mathcal{H}}(\mathcal{A} \otimes \mathcal{B}), \emptyset)^{\perp s}.$$

where $\mathcal{A}^{\perp s}$ is a refinement of the orthogonal of $\mathcal{A}$ including only simple (co-)terms. The operation $Cons_{\mathcal{H}}(\mathcal{A} \otimes \mathcal{B})$ gives a set:

$$Cons_{\mathcal{H}}(\mathcal{A} \otimes \mathcal{B}) \triangleq \{(V_1, V_2) | V_1 \in \mathcal{A}, V_2 \in \mathcal{B}\}.$$

We now give an $\mathcal{H}$ such that $\mathcal{R}(\langle\!\langle \mathsf{F}(\overrightarrow{A})\rangle\!\rangle_{\mathcal{H}}) = \mathcal{H}(\mathsf{F}(\overrightarrow{A}))$, by well-founded recursion over hybrid types based on their dependencies. The syntactic well-formedness checks for (co-)data declarations ensure well-foundedness of the dependency order of hybrid types, such that changing $\mathcal{H}$ for any $h' > h$ does not change $\langle\!\langle h\rangle\!\rangle_{\mathcal{H}}$ (this condition is an example of "contractive functions" in the general framework of *Complete Ordered Families of Equivalence* [3]).

However, we must take care with size types. We use $\mathcal{H}$ to interpret type variables, as well as giving the meaning of 0, $M + 1$, and ∞. We interpret sizes (both Ix and Ord) as countable ordinals, allowing for induction. It might seem that we could just set $\mathcal{H}(0) = 0$, but this does not work in general. The problem is that we might have a sequent, $c : \Gamma \vdash_{i<0} \Delta$, where we still want to prove that c is strongly normalizing. We therefore need to have *some* valid interpretation function $\mathcal{H}$ where $\mathcal{H}(i) < \mathcal{H}(0)$. *The meaning of zero might not be zero.* Luckily, we can come up with such an $\mathcal{H}$ by examining the length of Θ. Although case analysis may introduce new bindings like $i < j$, for which it is impossible to ensure the existence of a value less than $\mathcal{H}(j)$, this is not a problem since we only reduce inside case abstractions when all kinds are inhabited due to the reachability caveat. The meaning of a set of declared type constructors, $\mathcal{H} \in [\![\mathcal{F}]\!]$, includes partial functions from hybrid types to semantic types (reducibility candidates, countable ordinals, and functions between them) such that (1) given any $\mathsf{F} \in \mathcal{F}$ and appropriate arguments $\overrightarrow{\mathcal{A}}$ we have that $\langle\!\langle \mathsf{F}(\overrightarrow{\mathcal{A}})\rangle\!\rangle_{\mathcal{H}} \sqsubseteq \mathcal{H}(\mathsf{F}(\overrightarrow{\mathcal{A}}))$,

(2) $\mathcal{H}(0)$ and $\mathcal{H}(\infty)$ are ordinals with $\mathcal{H}(\infty)$ as a limit ordinal and $\mathcal{H}(0) < \mathcal{H}(\infty)$, and (3) $\mathcal{H}(+1(\mathcal{M})) > \mathcal{M}$ for any ordinal $\mathcal{M}$.

Lemma 2. *For any well-formed $\mathcal{F}$, there exists a $\mathcal{H} \in [\![\mathcal{F}]\!]$.*

Another source of complexity is that some syntactic types matter computationally, in the case of indices. Reduction rules (e.g for $\mathsf{Inflate}$) depend on the syntactic form of the index. We resolve this by interpreting kinds not just as sets of semantic objects, but also as logic relations between syntactic and semantic types [8]. Because kinds can depend on types, the meaning of semantic types and kinds (as sets) are intertwined.

$$[\![a]\!]_{\mathcal{H}} \triangleq \mathcal{H}(a) \qquad [\![\lambda a : k.B]\!]_{\mathcal{H}} \triangleq \lambda\mathcal{A} \in [\![k]\!].[\![B]\!]_{\mathcal{H}\{\mathcal{A}/a\}}$$
$$[\![\mathsf{F}(\overrightarrow{A})]\!]_{\mathcal{H}} \triangleq \mathcal{H}(\mathsf{F}(\overrightarrow{[\![A]\!]_{\mathcal{H}}})) \qquad [\![A\ B]\!]_{\mathcal{H}} \triangleq [\![A]\!]_{\mathcal{H}}([\![B]\!]_{\mathcal{H}})$$
$$[\![\star]\!]_{\mathcal{H}} \triangleq CR \qquad [\![\mathsf{Ix}]\!]_{\mathcal{H}} \triangleq \{\mathcal{H}(0)\} \cup \{\mathcal{H}(+1(\mathcal{M})) | \mathcal{M} \in [\![\mathsf{Ix}]\!]\}$$
$$[\![\mathsf{Ord}]\!]_{\mathcal{H}} \triangleq \mathbb{O} \qquad [\![< N]\!]_{\mathcal{H}} \triangleq \{\mathcal{M} \in \mathbb{O} | [\![N]\!]_{\mathcal{H}} \leq \mathcal{M}\}$$

$$\begin{aligned}
A\ [\![\star]\!]_{\mathcal{H}}\ \mathcal{A} &\iff true \\
A\ [\![k_1 \to k_2]\!]_{\mathcal{H}}\ \mathcal{A} &\iff \forall B [\![k_1]\!]_{\mathcal{H}} \mathcal{B}.A\ B\ [\![k_2]\!]_{\mathcal{H}}\ \mathcal{A}(\mathcal{B}) \\
M\ [\![\mathsf{Ix}]\!]_{\mathcal{H}}\ \mathcal{M} &\iff [\![M]\!]_{\mathcal{H}} \sim_{\mathcal{H}} \mathcal{M} \\
M\ [\![\mathsf{Ord}]\!]_{\mathcal{H}}\ \mathcal{M} &\iff \exists M \twoheadrightarrow M'.[\![M']\!]_{\mathcal{H}} \leq \mathcal{M} \\
M\ [\![< N]\!]_{\mathcal{H}}\ \mathcal{M} &\iff \exists M \twoheadrightarrow M'.[\![M']\!]_{\mathcal{H}} \leq \mathcal{M} < [\![N]\!]_{\mathcal{H}}
\end{aligned}$$

Note that $[\![\blacksquare]\!]$ contains all these semantic kinds. Furthermore, $[\![\square]\!]$ adds the requirement that the relation between syntactic and semantic types is backward closed under syntactic β reduction of the λ-calculus: if $\kappa \in [\![\square]\!]$, $A \twoheadrightarrow_{\beta} B$, and $B\ \kappa\ \mathcal{B}$, then $A\ \kappa\ \mathcal{B}$.

We interpret typing environments as sets of substitutions.

$$\begin{aligned}
\gamma \in [\![\Gamma]\!]_{\mathcal{H}} &\iff \forall x : A \in \Gamma.x\{\gamma\} \in Val([\![A]\!]_{\mathcal{H}}) \\
\delta \in [\![\Delta]\!]_{\mathcal{H}} &\iff \forall \alpha : A \in \Delta.\alpha\{\delta\} \in Val([\![A]\!]_{\mathcal{H}})
\end{aligned}$$

Kinding environments are interpreted as relations between syntactic substitutions and semantic interpretations.

$$\begin{aligned}
\theta[\![\epsilon]\!]_{\mathcal{H}}\mathcal{I} &\iff \forall h.\mathcal{H}(h) = \mathcal{I}(h) \\
\theta[\![\Theta', a : k]\!]_{\mathcal{H}}\mathcal{I} &\iff \exists \theta'[\![\Theta']\!]_{\mathcal{H}}\mathcal{I}'.a\{\theta\}[\![k]\!]_{\mathcal{I}}\mathcal{I}(a) \\
&\qquad \wedge (\forall b \neq a.b\{\theta\} = b\{\theta'\}) \\
&\qquad \wedge (\forall h \neq a.\mathcal{I}(h) = \mathcal{I}'(h))
\end{aligned}$$

The main soundness property is that if $c : \Gamma \vdash_{\Theta} \Delta$ and $(\Gamma \vdash_{\Theta} \Delta)\,\mathbf{seq}$ are derivable, then for any $\mathcal{H} \in [\![\mathcal{F}]\!]$, where $\mathcal{H}$ assigns a big enough meaning to 0, then $[\![c : \Gamma \vdash_{\Theta} \Delta]\!]_{\mathcal{H}}$ as given in Figure 1. Further, because (co-)variables inhabit every reducibility candidate, $[\![c : (\Gamma \vdash_{\Theta} \Delta)]\!]_{\mathcal{H}}$ entails that c is strongly normalizing. Additionally, if $Erase(c) \to c'$ then there must be a command c'' such that $c \to c''$ and $c' = Erase(c'')$ since the type-erased reduction rules are more limited than the ones for non-erased commands. Thus, if a typed command c is strongly normalizing, then $Erase(c)$ must be as well.

Theorem 1. *If $c : \Gamma \vdash_{\Theta} \Delta$ and $(\Gamma \vdash_{\Theta} \Delta)\,\mathbf{seq}$, then c is strongly normalizing in the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus. Furthermore, $Erase(c)$ is strongly normalizing in the type-erased $\mu\tilde{\mu}_{\mathcal{S}}$-calculus.*

B. Pre-types

A pre-type is a pair of a set of terms and a set of co-terms. If $\mathcal{A}$ is a pre-type, we write $v \in \mathcal{A}$ or $e \in \mathcal{A}$ to indicate that a given (co-)term is an element of that pre-type.

Since we are interested in strong normalization, we carve out the strongly normalizing commands and (co-)terms. The set $\perp\!\!\!\perp$ of commands consists of all those commands which are strongly normalizing. $\perp\!\!\!\perp$ is closed under reduction. The pre-type $\mathcal{W}$ of well-behaved (co-)terms is defined as containing all the strongly normalizing (co-)terms. Note that since all reducts of strongly normalizing commands and (co-)terms are themselves strongly normalizing, $\perp\!\!\!\perp$ and $\mathcal{W}$ are forward closed: if $c \in \perp\!\!\!\perp$ and $c \to c'$ then $c' \in \perp\!\!\!\perp$, if $v \in \mathcal{W}$ and $v \to v'$ then $v' \in \mathcal{W}$, and if $e \in \mathcal{W}$ and $e \to e'$ then $e' \in \mathcal{W}$.

There are two fundamental orderings of pre-types. Given two pre-types $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ *refines* $\mathcal{B}$, written $\mathcal{A} \sqsubseteq \mathcal{B}$, if and only if

$$\begin{aligned}
v \in \mathcal{A} &\implies v \in \mathcal{B} \\
e \in \mathcal{A} &\implies e \in \mathcal{B}
\end{aligned}$$

By contrast, $\mathcal{A}$ is a *subtype* of $\mathcal{B}$, written $\mathcal{A} \leq \mathcal{B}$, if and only if

$$\begin{aligned}
v \in \mathcal{A} &\implies v \in \mathcal{B} \\
e \in \mathcal{B} &\implies e \in \mathcal{A}
\end{aligned}$$

Note that these two orderings form a complete lattice on the set of strongly-normalizing pre-types. In terms of refinement, $(\emptyset, \emptyset)$ is the smallest element that refines every pre-type, and $\mathcal{W}$ serves as a largest element that every strongly-normalizing pre-type refines. In terms of subtyping, $(\emptyset, \{e \in \mathcal{W}\})$ is the smallest element that is a sub-type of every pre-type, and $(\{v \in \mathcal{W}\}, \emptyset)$ is the largest element that is a super-type of every pre-type. In general, we use square operations to refer to operations of the refinement lattice, and triangular operations to refer to the subtyping lattice. In particular given the semantic types $\mathcal{A} = (\mathcal{A}^+, \mathcal{A}^-)$ and $\mathcal{B} = (\mathcal{B}^+, \mathcal{B}^-)$, where $\mathcal{A}^+$ and $\mathcal{B}^+$ contain the terms of $\mathcal{A}$ and $\mathcal{B}$ and dually for $\mathcal{A}^-, \mathcal{B}^-$, we have the joins and meets of both orders:

$$\begin{aligned}
(\mathcal{A}^+, \mathcal{A}^-) \sqcup (\mathcal{B}^+, \mathcal{B}^-) &\triangleq (\mathcal{A}^+ \cup \mathcal{B}^+, \mathcal{A}^- \cup \mathcal{B}^-) \\
(\mathcal{A}^+, \mathcal{A}^-) \sqcap (\mathcal{B}^+, \mathcal{B}^-) &\triangleq (\mathcal{A}^+ \cap \mathcal{B}^+, \mathcal{A}^- \cap \mathcal{B}^-) \\
(\mathcal{A}^+, \mathcal{A}^-) \vee (\mathcal{B}^+, \mathcal{B}^-) &\triangleq (\mathcal{A}^+ \cup \mathcal{B}^+, \mathcal{A}^- \cap \mathcal{B}^-) \\
(\mathcal{A}^+, \mathcal{A}^-) \wedge (\mathcal{B}^+, \mathcal{B}^-) &\triangleq (\mathcal{A}^+ \cap \mathcal{B}^+, \mathcal{A}^- \cup \mathcal{B}^-)
\end{aligned}$$

Lemma 3 (Pre-type lattice)**.** *Each of the $\leq$ and $\sqsubseteq$ orderings on the set of pre-types form a complete lattice: that is, they have all joins and meets.*

Proof. The set of subsets of a set is a complete lattice ordered by $\subseteq$ with the usual $\bigcup$ and $\bigcap$ operations. Further, the dual of a complete lattice is itself a complete lattice, and the product of two complete lattices is a complete lattice. The case of $\sqsubseteq$ is the product of the two subset lattices; the case of $\leq$ is the product of the two subset lattices where one is dualized. $\square$

Lemma 4. *$\sqcup$ is monotonic (in both arguments) with respect to $\leq$ and is commutative and associative.*

Proof. That $\sqcup$ is commutative and associative follows immediately from its definition. The interesting fact is that, for any pre-types $\mathcal{A} = (\mathcal{A}^+, \mathcal{A}^-)$, $\mathcal{B} = (\mathcal{B}^+, \mathcal{B}^-)$, and $\mathcal{C} = (\mathcal{C}^+, \mathcal{C}^-)$ if $\mathcal{A} \leq \mathcal{B}$ then we have

$$\begin{aligned}
\mathcal{A} \sqcup \mathcal{C} &= (\mathcal{A}^+ \cup \mathcal{C}^+, \mathcal{A}^- \cup \mathcal{C}^-) \\
&\leq (\mathcal{B}^+ \cup \mathcal{C}^+, \mathcal{B}^+ \cup \mathcal{C}^-) \\
&= \mathcal{B} \sqcup \mathcal{C}
\end{aligned}$$

since $\mathcal{A}^+ \subseteq \mathcal{B}^+$ means $\mathcal{A}^+ \cup \mathcal{C}^+ \subseteq \mathcal{B}^+ \cup \mathcal{C}^+$ and $\mathcal{B}^- \subseteq \mathcal{A}^-$ means $\mathcal{B}^- \cup \mathcal{C}^- \subseteq \mathcal{A}^- \cup \mathcal{C}^-$. $\square$

The most basic operation on pre-types is the orthogonal $A^{\perp}$:

$$\begin{aligned}
v \in \mathcal{A}^{\perp} &\iff v \in \mathcal{W} \wedge \forall e \in \mathcal{A}.\langle v \| e \rangle \in \perp\!\!\!\perp \\
e \in \mathcal{A}^{\perp} &\iff e \in \mathcal{W} \wedge \forall v \in \mathcal{A}.\langle v \| e \rangle \in \perp\!\!\!\perp
\end{aligned}$$

We say that a pre-type $\mathcal{A}$ is *orthogonally sound* if and only if $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$. In other words, for all $\mathcal{A} \sqsubseteq \mathcal{W}$ and for all $v, e \in \mathcal{A}$, $\langle v \| e \rangle \in \perp\!\!\!\perp$. Furthermore, a pre-type $\mathcal{A}$ is *orthogonally complete*

$$[\![c : \Gamma \vdash_\Theta \Delta]\!]_\mathcal{H} \Leftrightarrow \forall\theta[\![\Theta]\!]_\mathcal{H}\mathcal{I}.\forall\gamma \in [\![\Gamma]\!]_\mathcal{I}, \delta \in [\![\Delta]\!]_\mathcal{I}.c\{\theta,\gamma,\delta\} \in \perp\!\!\!\perp \qquad [\![\Gamma \vdash_\Theta v : A|\Delta]\!]_\mathcal{H} \Leftrightarrow \forall\theta[\![\Theta]\!]_\mathcal{H}\mathcal{I}.\forall\gamma \in [\![\Gamma]\!]_\mathcal{I}, \delta \in [\![\Delta]\!]_\mathcal{I}.v\{\theta,\gamma,\delta\} \in [\![A]\!]_\mathcal{I}$$

$$[\![\Theta \vdash A : k]\!]_\mathcal{H} \Leftrightarrow \forall\theta[\![\Theta]\!]_\mathcal{H}\mathcal{I}.A\{\theta\}[\![k]\!]_\mathcal{I}[\![A]\!]_\mathcal{I} \qquad [\![\Theta \vdash A = B : k]\!]_\mathcal{H} \Leftrightarrow \forall\mathcal{I} \in [\![\Theta]\!]_\mathcal{H}.[\![A]\!]_\mathcal{I} = [\![B]\!]_\mathcal{I} \in [\![k]\!]_\mathcal{I}$$

$$[\![\Theta \vdash k : s]\!]_\mathcal{H} \Leftrightarrow \forall\mathcal{I} \in [\![\Theta]\!]_\mathcal{H}.[\![k]\!]_\mathcal{I} \in [\![s]\!] \qquad [\![(\vdash)\,\mathbf{seq}\,]\!]_\mathcal{H} \Leftrightarrow true \qquad [\![(\vdash_{\Theta,a:k})\,\mathbf{seq}\,]\!]_\mathcal{H} \Leftrightarrow [\![(\vdash_\Theta)\,\mathbf{seq}\,]\!] \wedge [\![\Theta \vdash k : \blacksquare]\!]_\mathcal{H}$$

$$[\![(\Gamma \vdash_\Theta \Delta)\,\mathbf{seq}\,]\!]_\mathcal{H} \Leftrightarrow [\![(\vdash_\Theta)\,\mathbf{seq}\,]\!]_\mathcal{H} \wedge (\forall x : A \in \Gamma.[\![\Theta \vdash A : \star]\!]_\mathcal{H}) \wedge (\forall\alpha : A \in \Delta.[\![\Theta \vdash A : \star]\!]_\mathcal{H})$$

Figure 1. The semantic interpretation of sequents in the model.

if and only if $\mathcal{A}^\perp \sqsubseteq \mathcal{A}$. Therefore, an orthogonally sound and complete pre-type $\mathcal{A}$ is one such that $\mathcal{A} = \mathcal{A}^\perp$.

We can generalize the orthogonal operation to a general class of similar operations on pre-types that all share the same properties. We say that an operation on pre-types Op is a *negation operation inside* $\mathcal{D}$ if and only if $\mathcal{D}$ is a fixed pre-type and there exists predicates P and Q on term, co-term pairs such that:

$$v \in Op(\mathcal{A}) \iff v \in \mathcal{D} \wedge \forall e \in \mathcal{A}.P(v,e)$$
$$e \in Op(\mathcal{A}) \iff e \in \mathcal{D} \wedge \forall v \in \mathcal{A}.Q(v,e)$$

Furthermore, Op is a *symmetric negation operation inside* $\mathcal{D}$ if and only if for all $v, e \in \mathcal{D}$, $P(v,e) \iff Q(v,e)$. Note that orthogonality is a symmetric negation operation inside $\mathcal{W}$. It follows that all such negation operations enjoy the standard basic properties of orthogonality.

Lemma 5 (Monotonicity)**.** *For any negation operation Op inside $\mathcal{D}$, $\mathcal{A} \leq \mathcal{B}$ implies $Op(\mathcal{A}) \leq Op(\mathcal{B})$.*

Proof. Suppose $v \in Op(\mathcal{A})$, so we know that $v \in \mathcal{D}$ and for all $e \in \mathcal{A}$, $P(v,e)$. Then, given any $e \in \mathcal{B}$, we know that $e \in \mathcal{A}$ because $\mathcal{A} \leq \mathcal{B}$, and so $P(v,e)$. Therefore, $v \in Op(\mathcal{B})$ as well.

Suppose $e \in Op(\mathcal{B})$, so we know that $e \in \mathcal{D}$ and for all $e \in \mathcal{B}$, $Q(v,e)$. Then, given any $v \in \mathcal{A}$, we know that $v \in \mathcal{B}$ because $\mathcal{A} \leq \mathcal{B}$, and so $Q(v,e)$. Therefore, $e \in Op(\mathcal{A})$ as well. □

Lemma 6 (Contrapositive)**.** *For any negation operation Op inside $\mathcal{D}$, $\mathcal{A} \sqsubseteq \mathcal{B}$ implies $Op(\mathcal{B}) \sqsubseteq Op(\mathcal{A})$.*

Proof. Suppose $v \in Op(\mathcal{B})$, so we know that $v \in \mathcal{D}$ and for all $e \in \mathcal{B}$, $P(v,e)$. Then, given any $e \in \mathcal{A}$, we know that $e \in \mathcal{B}$ because $\mathcal{A} \sqsubseteq \mathcal{B}$, and so $P(v,e)$. Therefore, $v \in Op(\mathcal{B})$ as well.

Suppose $e \in Op(\mathcal{B})$, so we know that $e \in \mathcal{D}$ and for all $v \in \mathcal{B}$, $Q(v,e)$. Then, given any $v \in \mathcal{A}$, we know that $v \in \mathcal{B}$ because $\mathcal{A} \sqsubseteq \mathcal{B}$, and so $Q(v,e)$. Therefore, $e \in Op(\mathcal{B})$ as well. □

Lemma 7 (Double Negation Introduction)**.** *For any symmetric negation operation Op inside $\mathcal{D}$, $\mathcal{A} \sqsubseteq \mathcal{D}$ implies $\mathcal{A} \sqsubseteq Op(Op(\mathcal{A}))$.*

Proof. For any $v \in \mathcal{A}$ and $e \in Op(\mathcal{A})$, $Q(v,e)$ by definition of $Op(\mathcal{A})$, so $P(v,e)$ as well since Op is symmetric. Therefore, $\mathcal{A} \sqsubseteq \mathcal{D}$ implies that $v \in \mathcal{D}$, so that $v \in Op(Op(\mathcal{A}))$. The case of $e \in \mathcal{A}$ implies $e \in Op(Op(\mathcal{A}))$ is dual. □

Lemma 8 (Triple Negation Elimination)**.** *For any symmetric negation operation Op inside $\mathcal{D}$, $\mathcal{A} \sqsubseteq \mathcal{D}$ implies $Op(Op(Op(\mathcal{A}))) = Op(\mathcal{A})$.*

Proof. Note that $Op(\mathcal{A}) \sqsubseteq \mathcal{D}$ because Op is a negation operation inside $\mathcal{D}$. Therefore, by double negation introduction (Lemma 7), $Op(\mathcal{A}) \sqsubseteq Op(Op(Op(\mathcal{A})))$. Additionally, because $\mathcal{A} \sqsubseteq \mathcal{D}$, we know that $\mathcal{A} \sqsubseteq Op(Op(\mathcal{A}))$ by double negation introduction (Lemma 7), and so $Op(Op(Op(\mathcal{A}))) \sqsubseteq Op(\mathcal{A})$ by contrapositive (Lemma 6). Therefore, $Op(Op(Op(\mathcal{A}))) = Op(\mathcal{A})$. □

We can rephrase the pre-type $\mathcal{W}$ of well-behaved (co-)terms solely in terms of orthogonality and (co-)variables. This ensures to us that any other pre-type $\mathcal{A} \sqsubseteq \mathcal{W}$ which is orthogonally complete must contain (co-)variables.

Lemma 9. *1. v is strongly normalizing iff $\langle v\|\alpha\rangle$ is.*
2. e is strongly normalizing iff $\langle x\|e\rangle$ is.

Proof. 1. Since v is a sub-term of $\langle v\|\alpha\rangle$, strong normalization of $\langle v\|\alpha\rangle$ implies strong normalization of v.
Going the other way, we show that every reduction of $\langle v\|\alpha\rangle$, except for possibly one top-level μ_E reduction, can be traced by v as well. We proceed to show that $\langle v\|\alpha\rangle$ is strongly normalizing because all of its reducts are by well-founded induction on $|v|$:

- Suppose $v = \mu\beta.c$, so that we have the top-level μ_E reduction:
$$\langle \mu\beta.c\|\alpha\rangle \to_{\mu_E} c\{\alpha/\beta\}$$
Furthermore, we know $\mu\alpha.c\{\alpha/\beta\}$ is strongly normalizing since it is α-equivalent to the strongly normalizing $\mu\beta.c$, which means that $c\{\alpha/\beta\}$ is also strongly normalizing since it is a sub-command of $\mu\alpha.c\{\alpha/\beta\}$.
- Suppose we have some other reduction internal to v, so that:
$$\langle v\|\alpha\rangle \to \langle v'\|\alpha\rangle$$
Then we know that $v \to v'$ so $|v'| < |v|$. Therefore, by the inductive hypothesis, we get that $\langle v'\|\alpha\rangle$ is strongly normalizing.
Since every reduct of $\langle v\|\alpha\rangle$ is strongly normalizing, then $\langle v\|\alpha\rangle$ is also strongly normalizing.

2. Analogous to the above by duality. □

Corollary 1. $\mathcal{W} = Var^\perp$.

Proof. Note that $Var^\perp$ is the semantic type:

$$v \in Var^\perp \iff v \in \mathcal{W} \wedge \forall\alpha \in Var.\langle v\|\alpha\rangle \in \perp\!\!\!\perp$$
$$e \in Var^\perp \iff e \in \mathcal{W} \wedge \forall x \in Var.\langle x\|e\rangle \in \perp\!\!\!\perp$$

And so $v, e \in \mathcal{W}$ if and only if $v, e \in Var^\perp$ by the above Lemma 9. □

Corollary 2. *If $\mathcal{A}^\perp \sqsubseteq \mathcal{A} \sqsubseteq \mathcal{W}$ then $Var \sqsubseteq \mathcal{A}$.*

Proof. Using the above Corollary 1, we conclude by double negation introduction (Lemma 7) and contrapositive (Lemma 8):

$$Var \sqsubseteq Var^{\perp\perp} = \mathcal{W}^\perp \sqsubseteq \mathcal{A}^\perp \sqsubseteq A$$ □

C. Head Reduction

There are two important properties for the chosen strategy $\mathcal{S}$ needed for the proof of strong normalization. First, we say a strategy *stable* if and only if:

1. (co-)values are closed under reduction and substitution, and
2. non-(co-)values are closed under substitution and ς reduction.

Second, we say a strategy is *focalizing* if and only if all:

1. (co-)variables,
2. structures built from (co-)values, and
3. case abstractions (recursive or non-recursive)

are considered (co-)values. The focalizing property of strategies corresponds to focalization in logic [7]—each criterion for a focalizing strategy comes from an inference rule for typing a (co-)value in focus. For the remainder of this proof, we assume that the chosen strategy $\mathcal{S}$ is both stable and focalizing.

The head reduction relation describes only those reductions that happen at the top of a command and is given in Figure 2. These reductions are *charged*. Neutrally charged reductions $\mapsto_0$ require cooperation of the term and co-term, like in the β rules. Positively charged reductions $\mapsto_+$ allow the term to take over the command in order to simplify itself. Negatively charged reductions $\mapsto_-$ are allow the co-term to take over the command. There are several useful facts that are immediately apparent about this definition of head reduction.

1. Every step of head reduction is simulated by several steps of the general reduction theory: $c \mapsto_{+,0,-} c'$ implies that $c \twoheadrightarrow c'$.
2. Head reduction only occurs when one side of the command is a (co-)value: if $\langle v \| e\rangle \mapsto_+$ then e is a co-value, if $\langle v \| e\rangle \mapsto_-$ then v is a value, and if $\langle v \| e\rangle \mapsto_0$ then both v and e are (co-)values. This last point about neutral head reductions implying both sides of the command is a (co-)value follows from the assumption that the chosen strategy $\mathcal{S}$ is focalizing.
3. When taken on their own, each of the charged head reduction relations, $\mapsto_0$, $\mapsto_+$, and $\mapsto_-$ are deterministic regardless of the chosen strategy, although the combined $\mapsto_{+/-}$ head reduction relation may be non-deterministic. Additionally, the combined $\mapsto_{+,0}$ and $\mapsto_{-,0}$ reduction relations are deterministic as well.

Furthermore, head reduction steps commute with the internal reductions inside either side of the command.

Lemma 10. *1. If $v \to v'$ and $\langle v \| e\rangle \mapsto_0 c$ then there is a c' such that $\langle v' \| e\rangle \mapsto_0 c'$ and $c \twoheadrightarrow c'$.*
2. If $e \to e'$ and $\langle v \| e\rangle \mapsto_0 c$ then there is a c' such that $\langle v \| e'\rangle \mapsto_0 c'$ and $c \twoheadrightarrow c'$.
3. If $v \to v'$ and $\langle v \| e\rangle \mapsto_- c$ then there is a c' such that $\langle v' \| e\rangle \mapsto_- c'$ and $c \twoheadrightarrow c'$.
4. If $e \to e'$ and $\langle v \| e\rangle \mapsto_+ c$ then there is a c' such that $\langle v \| e'\rangle \mapsto_- c$ and $c \twoheadrightarrow c'$.
5. If $v \to v'$ then either
(a) $\forall e, c$ such that $\langle v \| e\rangle \mapsto_+ c$ we have $c \twoheadrightarrow \langle v' \| e\rangle$, or
(b) $\forall e, c$ such that $\langle v \| e\rangle \mapsto_+ c$ there exists a c' such that $\langle v' \| e\rangle \mapsto_+ c'$ and $c \twoheadrightarrow c'$.
6. If $e \to e'$ then either
(a) $\forall v, c$ such that $\langle v \| e\rangle \mapsto_- c$ we have $c \twoheadrightarrow \langle v \| e'\rangle$ or
(b) $\forall v, c$ such that $\langle v \| e\rangle \mapsto_- c$ there exists a c' such that $\langle v \| e'\rangle \mapsto_- c'$ and $c \twoheadrightarrow c'$.

Proof. By cases on the possible reductions. Note that for statements 1-4, there are no critical pairs between neutral head reductions and general reductions, because (co-)values are closed under reduction by the stability of $\mathcal{S}$. Additionally, for the other statements, the fact that (co-)values are closed under reduction cuts out many critical pairs. For statement 5, we illustrate the remaining interesting critical pairs:

- $\mu\alpha.\langle v \| \alpha\rangle \to v$. Note that this is equivalent in the only possible head reduction in the command $\langle \mu\alpha.\langle v \| \alpha\rangle \| E\rangle \mapsto_+ \langle v \| E\rangle$, so we have case (a).
- $C^{\mathsf{K}}_{\varsigma}[v] \to \mu\alpha.\langle v \| \tilde{\mu}x.\langle C^{\mathsf{K}}_{\varsigma}[x] \| \alpha\rangle\rangle$. Given that
$$\langle C^{\mathsf{K}}_{\varsigma}[v] \| E\rangle \mapsto_+ \langle v \| \tilde{\mu}x.\langle C^{\mathsf{K}}_{\varsigma}[x] \| E\rangle\rangle$$
we have case (b) where c' is equal to the same $\langle v \| \tilde{\mu}x.\langle C^{\mathsf{K}}_{\varsigma}[x] \| E\rangle\rangle$ and
$$\langle \mu\alpha.\langle v \| \tilde{\mu}x.\langle C^{\mathsf{K}}_{\varsigma}[x] \| \alpha\rangle\rangle \| E\rangle \mapsto_+ \langle v \| \tilde{\mu}x.\langle C^{\mathsf{K}}_{\varsigma}[x] \| E\rangle\rangle$$
- $C^{\mathsf{K}}_{\varsigma}[e] \to \mu\alpha.\langle \mu\beta.\langle C^{\mathsf{K}}_{\varsigma}[\beta] \| \alpha\rangle \| e\rangle$. Analogous to a previous case by duality.
- $C^{\mathsf{K}}_{\varsigma}[v] \to C^{\mathsf{K}}_{\varsigma}[V]$. Given that
$$\langle C^{\mathsf{K}}_{\varsigma}[v] \| E\rangle \mapsto_+ \langle v \| \tilde{\mu}x.\langle C^{\mathsf{K}}_{\varsigma}[x] \| E\rangle\rangle$$
we have case (a) where
$$\langle v \| \tilde{\mu}x.\langle C^{\mathsf{K}}_{\varsigma}[x] \| E\rangle\rangle \to \langle V \| \tilde{\mu}x.\langle C^{\mathsf{K}}_{\varsigma}[x] \| E\rangle\rangle \to \langle C^{\mathsf{K}}_{\varsigma}[V] \| E\rangle$$
- $C^{\mathsf{K}}_{\varsigma}[e] \to C^{\mathsf{K}}_{\varsigma}[E]$. Analogous to a previous case by duality.
- $V\{N/i\} \to \mu(\mathsf{Rise}^{i<N}[\alpha](x).c\{M/j, V/x\})$ where $V = \mu(\mathsf{Rise}^{j<i}[\alpha](x).c)$. Note that this is equivalent to the only possible head reduction in the command $\langle V\{N/i\} \| E\rangle$, so we have case (a).

For statement 6, commutation follows analogously by duality. □

A term is *simple* if it never causes a positive head reduction. A co-term is simple if it never causes a negative head reduction. A pre-type is simple if all its (co-)terms are. Of note, observe that because our chosen strategy is assumed to be focalizing, all simple (co-)terms are (co-)values: a simple (co-)term can either take a neutral head reduction step, in which case it must be a (co-)value, or it is is a variable and doesn't actively participate in any reduction, so it must be a (co-)value by the assumption that the strategy is focalizing. Furthermore, the set of simple (co-)terms is closed under reduction because (co-)values are closed under reduction (since the strategy is stable). Also of note is the fact that all non-simple (co-)terms (that is, the ones which can cause a positive or negative head reduction) can never be part of a neutral reduction.

We define the operation $Simp(\mathcal{A})$ as containing all the (co-)terms of $\mathcal{A}$ which are simple. There is a simplified version of the orthogonality operation, $(-)^{\perp s}$, that operates in the simplified version of well-behaved (co-)terms $\mathcal{W}$. More specifically, the $(-)^{\perp s}$ is defined as:
$$\mathcal{A}^{\perp s} \triangleq Simp(\mathcal{A}^{\perp})$$
Note that $(-)^{\perp s}$ is a symmetric negation operation inside $Simp(\mathcal{W})$, so all the basic properties of monotonicity, contrapositive, double negation introduction, and triple negation elimination apply.

D. Reducibility Candidates

To define the the set of reducibility candidates, we have to determine the pre-types are sufficiently saturated so that they contain enough (co-)terms such that the general typing rules are sound, without invalidating the Cut rule. For example, we need to be sure that reducibility candidates contain the necessary μ- and $\tilde{\mu}$-abstractions according to Act and $CoAct$. We characterize this saturation in terms of head reduction, using the $Head(-)$ operation on pre-types defined as:
$$v \in Head(\mathcal{A}) \iff v \in \mathcal{W} \wedge \forall E \in \mathcal{A}.\langle v \| E\rangle \mapsto_+ c \in \perp\!\!\!\perp$$
$$e \in Head(\mathcal{A}) \iff e \in \mathcal{W} \wedge \forall V \in \mathcal{A}.\langle V \| e\rangle \mapsto_- c \in \perp\!\!\!\perp$$
Note that $Head(-)$ is a (non-symmetric) negation operation inside$\mathcal{W}$, so the monotonicity and contrapositive properties apply.

We now define reducibility candidates as all orthogonally sound and complete pre-types that contain their own $Head$:

$$
\begin{array}{c}
\langle \mu\alpha.c \| E\rangle \mapsto_{+} c\{E/\alpha\} \qquad \langle V \| \tilde{\mu}x.c\rangle \mapsto_{-} c\{V/x\} \\
\langle \mathsf{K}^{\vec{B}}(\vec{E},\vec{V}) \| \tilde{\mu}[\mathsf{K}^{\vec{b}}(\vec{\alpha},\vec{x}).c] | \dots\rangle \mapsto_{0} c\{\overrightarrow{B/b},\overrightarrow{E/\alpha},\overrightarrow{V/x}\} \qquad \langle \mu(\mathsf{H}^{\vec{b}}[\vec{x},\vec{\alpha}].c | \dots) \| \mathsf{H}^{\mathsf{B}}[\vec{V},\vec{E}]\rangle \mapsto_{0} c\{\overrightarrow{B/b},\overrightarrow{V/x},\overrightarrow{E/\alpha}\} \\
\langle C^{\mathsf{K}}_{\varsigma}[v] \| E\rangle \mapsto_{+} \langle \mu\alpha.\langle v \| \tilde{\mu}y.\langle C^{\mathsf{K}}_{\varsigma} \| \alpha\rangle\rangle \| E\rangle \qquad \langle V \| C^{\mathsf{H}}_{\varsigma}[v]\rangle \mapsto_{-} \langle V \| \tilde{\mu}x.\langle v \| \tilde{\mu}y.\langle x \| C^{\mathsf{H}}_{\varsigma}[y]\rangle\rangle\rangle \qquad \textbf{where}\, v \notin Value \\
\langle C^{\mathsf{K}}_{\varsigma}[e] \| E\rangle \mapsto_{+} \langle \mu\alpha.\langle \mu\beta.\langle C^{\mathsf{K}}_{\varsigma}[\beta] \| \alpha\rangle \| e\rangle \| E\rangle \qquad \langle V \| C^{\mathsf{H}}_{\varsigma}[e]\rangle \mapsto_{-} \langle V \| \tilde{\mu}x.\langle \mu\beta.\langle x \| C^{\mathsf{H}}_{\varsigma}[\beta]\rangle \| e\rangle\rangle \qquad \textbf{where}\, e \notin CoValue \\
\langle \mu(\mathsf{Rise}^{j<N}[\alpha](x).c) \| E\rangle \mapsto_{+} \langle \mu(\mathsf{Rise}^{i<N}[\alpha].c\{i/j, \mu(\mathsf{Rise}^{j<i}[\alpha](x).c)/x\}) \| E\rangle \\
\langle V \| \tilde{\mu}[\mathsf{Fall}^{j<N}(x)[\alpha].c]\rangle \mapsto_{-} \langle V \| \tilde{\mu}[\mathsf{Fall}^{i<N}(x).c\{i/j, \tilde{\mu}[\mathsf{Fall}^{j<i}(x)[\alpha].c]\}]\rangle \\
\langle V \| \mathsf{Up}^{0}[E]\rangle \mapsto_{0} c_0\{E/\alpha\} \qquad \langle V \| \mathsf{Up}^{M+1}[E]\rangle \mapsto_{0} \langle \mu\beta.\langle V \| \mathsf{Up}^{M}[\beta]\rangle \| \tilde{\mu}x.c_1\{M/j, E/\alpha\}\rangle \qquad \textbf{where}\, V = \mu(\mathsf{Up}^{0}[\alpha].c_0 | \mathsf{Up}^{j+1}[\alpha](x).c_1) \\
\langle \mathsf{Down}^{0}(V) \| E\rangle \mapsto_{0} c_0\{V/x\} \qquad \langle \mathsf{Down}^{M+1}(V) \| E\rangle \mapsto_{0} \langle \mu\alpha.c_1\{M/j, V/x\} \| \tilde{\mu}y.\langle \mathsf{Down}^{M}(y) \| E\rangle\rangle \qquad \textbf{where}\, E = \tilde{\mu}[\mathsf{Down}^{0}(x).c_0 | \mathsf{Down}^{j+1}(x)[\alpha].c_1]
\end{array}
$$

Figure 2. Head Reductions.

Definition 1 (Reducibility candidates)**.** *A semantic type, $\mathcal{A}$, is a* reducibility candidate *iff $\mathcal{A} = \mathcal{A}^{\perp}$ and $Head(\mathcal{A}) \sqsubseteq \mathcal{A}$. The set of reducibility candidates is written CR.*

We can show that a reducibility candidate must contain all the μ- and $\tilde{\mu}-$ abstractions that are well-behaved when paired with any of its (co-)values. This follows from the fact that the μ_E and $\tilde{\mu}_V$ reductions are non-neutrally charged head reductions.

Lemma 11 (Strong activation)**.** *1. If $\mathcal{A}$ is a reducibility candidate and for all $E \in \mathcal{A}$, $c\{E/\alpha\} \in \perp\!\!\!\perp$, then $\mu\alpha.c \in \mathcal{A}$.*
2. If $\mathcal{A}$ is a reducibility candidate and for all $V \in \mathcal{A}$, $c\{V/x\} \in \perp\!\!\!\perp$, then $\tilde{\mu}x.c \in \mathcal{A}$.

Proof. • Since $\mathcal{A}$ is a reducibility candidate, we know that $Head(\mathcal{A}) \sqsubseteq \mathcal{A}$. Observe that for all $E \in \mathcal{A}$,

$$\langle \mu\alpha.c \| E\rangle \mapsto c\{E/\alpha\} \in \perp\!\!\!\perp$$

by assumption. Therefore, $\mu\alpha.c \in Head(\mathcal{A}) \sqsubseteq \mathcal{A}$.
• Analogous to the previous statement by duality. □

Additionally, if a reducibility candidate contains all the structures built from (co-)values of other reducibility candidates, then it must contain the general constructs built from (co-)terms as well. This follows from the fact that all the lifting rules are implemented as non-neutrally charged head reductions.

Lemma 12 (Unfocalization)**.** *1. If $\mathcal{A}$,$\vec{\mathcal{B}}$,$\vec{\mathcal{C}}$ are reducibility candidates and for all $\overrightarrow{V \in \mathcal{B}}$,$\overrightarrow{E \in \mathcal{C}}$, $\mathsf{K}^{\vec{D}}(\vec{E},\vec{V}) \in \mathcal{A}$, then for all $\overrightarrow{v \in \mathcal{B}}$,$\overrightarrow{e \in \mathcal{C}}$, $\mathsf{K}^{\vec{D}}(\vec{e},\vec{v}) \in \mathcal{A}$.*
2. If $\mathcal{A}$,$\vec{\mathcal{B}}$,$\vec{\mathcal{C}}$ are reducibility candidates and for all $\overrightarrow{V \in \mathcal{B}}$,$\overrightarrow{E \in \mathcal{C}}$, $\mathsf{H}^{\vec{D}}[\vec{V},\vec{E}] \in \mathcal{A}$, then for all $\overrightarrow{v \in \mathcal{B}}$,$\overrightarrow{e \in \mathcal{C}}$, $\mathsf{H}^{\vec{D}}[\vec{v},\vec{e}] \in \mathcal{A}$.

Proof. • We proceed by right-to-left induction on the immediate non-(co-)value sub-(co-)terms $\vec{e}$, $\vec{v}$. Note that because $\mathcal{A}$ is a reducibility candidate, $Head(\mathcal{A}) \sqsubseteq \mathcal{A}$, so it suffices to show that $\mathsf{K}^{\vec{D}}(\vec{e},\vec{v}) \in Head(\mathcal{A})$.

- All of $\vec{e}$, $\vec{v}$ are (co-)values: $\mathsf{K}^{\vec{D}}(\vec{e},\vec{v}) \in \mathcal{A}$ by assumption.
- All of $\vec{e}$ are co-values, and v_i is the right-most non-value in $\vec{v} = \vec{v'}, v_i, \vec{V'}$: Observe that for any $E \in \mathcal{A}$, we have the head lifting reduction

  $$\langle \mathsf{K}^{\vec{D}}(\vec{e},\vec{v}) \| E\rangle \mapsto \langle v_i \| \tilde{\mu}x.\langle \mathsf{K}^{\vec{D}}(\vec{e},\vec{v'},x,\vec{V'}) \| E\rangle\rangle \in \perp\!\!\!\perp$$

  because $\tilde{\mu}x.\langle \mathsf{K}^{\vec{D}}(\vec{e},\vec{v'},x,\vec{V'}) \| E\rangle \in \mathcal{B}_i$ by strong activation (Lemma 11) and the inductive hypothesis. Therefore, $\mathsf{K}^{\vec{D}}(\vec{e},\vec{v}) \in Head(\mathcal{A})$.
- e_i is the right-most non-co-value in $\vec{e} = \vec{e'}, e_i, \vec{E'}$: Observe that for any $E \in \mathcal{A}$, we have the head lifting reduction

  $$\langle \mathsf{K}^{\vec{D}}(\vec{e},\vec{v}) \| E\rangle \mapsto \langle \mu\alpha.\langle \mathsf{K}^{\vec{D}}(\vec{E'},\alpha,\vec{e'},\vec{v}) \| E\rangle \| e_i\rangle \in \perp\!\!\!\perp$$

  because $\mu\alpha.\langle \mathsf{K}^{\vec{D}}(\vec{E'},\alpha,\vec{e'},\vec{v}) \| E\rangle$ by strong activation (Lemma 11) and the inductive hypothesis. Therefore, we have $\mathsf{K}^{\vec{D}}(\vec{e},\vec{v}) \in Head(\mathcal{A})$.

• Analogous to the previous statement by duality. □

We now show how to generate a full-fledged reducibility candidate from any simple core definition for a type using a variant of the symmetric candidates technique of Barbanera and Berardi [1]. We define the saturation function $Sat(\mathcal{A})$, which determines all the (co-)terms that are strongly normalizing with all the (co-)values of $\mathcal{A}$ either now, or one step by head reduction in the future:

$$
\begin{aligned}
v \in Sat(\mathcal{A}) \iff{}& v \in \mathcal{W} \\
&\wedge \forall E \in \mathcal{A}.\langle v \| E\rangle \in \perp\!\!\!\perp \vee \langle v \| E\rangle \mapsto_{+} c \in \perp\!\!\!\perp \\
e \in Sat(\mathcal{A}) \iff{}& e \in \mathcal{W} \\
&\wedge \forall V \in \mathcal{A}.\langle V \| e\rangle \in \perp\!\!\!\perp \vee \langle V \| e\rangle \mapsto_{-} c \in \perp\!\!\!\perp
\end{aligned}
$$

So starting with a seed $\mathcal{C}$, we can use $Sat(-)$, to grow a full-fledged reducibility candidate by iteratively saturating it, one $Step$ at a time:

$$Step_{\mathcal{C}}(\mathcal{A}) = \mathcal{C} \sqcup Sat(\mathcal{A})$$

This process eventually finishes, due to the fact that preserves the subtyping order of the lattice of pre-types (Lemma 3).

Corollary 3.

$$\mathcal{A} \leq \mathcal{B} \implies Step_{\mathcal{C}}(\mathcal{A}) \leq Step_{\mathcal{C}}(\mathcal{B})$$

Proof. By monotonicity (Lemma 5) because $Step_{\mathcal{C}}$ is a negation operation inside $\mathcal{W}$, for any pre-type $\mathcal{C}$. □

Corollary 4. *For any $\mathcal{C}$, there is a fixed point $\mathcal{A} = Step_{\mathcal{C}}(\mathcal{A})$.*

The fact that the fixed point to the $Step_{\mathcal{C}}$ function exists lets us define a function that saturates any pre-type $\mathcal{C}$.

Corollary 5. *There exists a function on pre-types $\mathcal{R}(-)$ such that $\mathcal{R}(\mathcal{C}) = Step_{\mathcal{C}}(\mathcal{R}(\mathcal{C}))$.*

Proof. By Corollary 4 there is a fixed to the function $Step_{\mathcal{C}}$, so we take $\mathcal{R}(\mathcal{C})$ to be the least one. □

Note that this $\mathcal{R}(-)$ operation gives us something that is *almost* a reducibility candidate. For any $Sat(\mathcal{A}) \sqsubseteq \mathcal{A} \sqsubseteq \mathcal{W}$ we have that:

$$Head(\mathcal{A}) \sqsubseteq Sat(\mathcal{A}) \sqsubseteq \mathcal{A}$$
$$\mathcal{A}^{\perp} \sqsubseteq Val(\mathcal{A})^{\perp} \sqsubseteq Sat(\mathcal{A}) \sqsubseteq \mathcal{A}$$

So all that is remaining is to show that, for certain well-behaved choices for $\mathcal{C}$, $\mathcal{R}(\mathcal{C}) \sqsubseteq \mathcal{R}(\mathcal{C})^{\perp}$. In particular, we will find that $\mathcal{R}(\mathcal{C})$ is guaranteed to be orthogonally sound, and thus a reducibility candidate, whenever $\mathcal{C}$ is in $TypeCore$.

Definition 2. *The set $TypeCore$ consists of pre-types $\mathcal{C}$ such that $\mathcal{C} \sqsubseteq \mathcal{C}^{\perp s}$ and $\mathcal{C}$ is forward closed.*

Note all pre-types $\mathcal{C}$ in $TypeCore$ are orthogonally sound because $\mathcal{C} \sqsubseteq \mathcal{C}^{\perp s} \sqsubseteq \mathcal{C}^{\perp}$. Furthermore, since $(-)^{\perp s}$ is itself a negation operation in $Simp(\mathcal{W})$, $\mathcal{C} \sqsubseteq \mathcal{C}^{\perp s} \sqsubseteq Simp(\mathcal{W})$ so it contains only simple (co-)terms, which must be (co-)values because the chosen strategy $\mathcal{S}$ is assumed to be focalizing.

Lemma 13 (Head orthogonality)**.** *Suppose that $\mathcal{A} \sqsubseteq \mathcal{W}$ is forward closed and for all $v, e \in \mathcal{A}$, $\langle v \| e \rangle \mapsto_{+,0,-} c$ implies $c \in \perp\!\!\!\perp$. Then $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$.*

Proof. Note that a command c is in $\perp\!\!\!\perp$, meaning it is strongly normalizing, if and only if all reducts of c are in $\perp\!\!\!\perp$. Since $\mathcal{A} \sqsubseteq \mathcal{W}$, every (co-)term in $\mathcal{A}$ is strongly normalizing, so let $|v|$ and $|e|$ be the lengths of the longest reduction sequence from any $v, e \in \mathcal{A}$, respectively. We now proceed to show that for any $v, e \in \mathcal{A}$, $\langle v \| e \rangle \in \perp\!\!\!\perp$ because all of its reducts are in $\perp\!\!\!\perp$, by induction on $|v| + |e|$.

- $\langle v \| e \rangle \mapsto_{+,0,-} c$: we know $c \in \perp\!\!\!\perp$ by assumption.
- $\langle v \| e \rangle \to \langle v' \| e \rangle$ because $v \to v'$: then $v' \in \mathcal{A}$ because $\mathcal{A}$ is forward closed, and $|v'| < |v|$. Furthermore, we have that all head reducts of $\langle v' \| e \rangle$ are in $\perp\!\!\!\perp$ by Lemma 10 and the fact that $\perp\!\!\!\perp$ is closed under reduction. Therefore, $\langle v' \| e \rangle \in \perp\!\!\!\perp$ by the inductive hypothesis.
- $\langle v \| e \rangle \to \langle v \| e' \rangle$ because $e \to e'$: analogous to the previous case by duality. ☐

Lemma 14. *If $v, e \in \mathcal{W}$ and $\langle v \| e \rangle \mapsto_0 c \in \perp\!\!\!\perp$, then $\langle v \| e \rangle \in \perp\!\!\!\perp$.*

Proof. Consider the pre-type $\mathcal{A}$ defined as

$$v' \in \mathcal{A} \iff v \twoheadrightarrow v'$$
$$e' \in \mathcal{A} \iff e \twoheadrightarrow e'$$

Note that v and e are in $\mathcal{A}$. Further, given any $v', e' \in \mathcal{A}$, $\langle v' \| e' \rangle \mapsto_0 c'$ such that $c \twoheadrightarrow c'$ by Lemma 10 and induction on their reduction paths, so that $c' \in \perp\!\!\!\perp$ because $\perp\!\!\!\perp$ is closed under reduction. Observe that $\mathcal{A} \sqsubseteq \mathcal{W}$ and is forward closed by definition. Finally, because v and e are simple, and the set of simple (co-)terms is forward closed, $\mathcal{A}$ is simple. From this, we know that for all $v', e' \in \mathcal{A}$, $\langle v' \| e' \rangle \mapsto_{0,+,-} c'$ implies that $c' \in \perp\!\!\!\perp$. Thus, $\mathcal{A} \sqsubseteq \mathcal{A}^{\perp}$ by Lemma 13 and so $\langle v \| e \rangle \in \perp\!\!\!\perp$. ☐

Lemma 15. *If $\mathcal{C} \in TypeCore$ and $\mathcal{A} = Step_{\mathcal{C}}(\mathcal{A})$, then for all $v, e \in \mathcal{A}$, $\langle v \| e \rangle \mapsto_{+,0,-} c$ implies $c \in \perp\!\!\!\perp$.*

Proof. Let $v, e \in \mathcal{A} = \mathcal{C} \sqcup Sat(\mathcal{A})$. By cases, we have:

- $V, E \in \mathcal{C}$: $V \perp\!\!\!\perp E$ by the assumption that $\mathcal{C} \in TypeCore$, so $\mathcal{C}$ is orthogonally sound.
- $V \in \mathcal{C}$, $e \in Sat(\mathcal{A})$: by $e \in Sat(\mathcal{A})$ and the fact that V is a value, we know that either $\langle V \| e \rangle \in \perp\!\!\!\perp$ or $\langle V \| e \rangle \mapsto_- c \in \perp\!\!\!\perp$ for some c. In the first case, every reduct of $\langle V \| e \rangle$ is in $\perp\!\!\!\perp$, including any head reductions, since $\perp\!\!\!\perp$ is forward closed. In the second case, we know V is simple by the assumption that $\mathcal{C} \in TypeCore$ and e is non-simple so it cannot take part of a neutral head reduction. This means $\langle V \| e \rangle \not\mapsto_{+/0}$ and $\langle V \| e \rangle \mapsto_- c'$ implies $c' \in \perp\!\!\!\perp$ because $\mapsto_-$ is deterministic so $c' = c \in \perp\!\!\!\perp$. Therefore, $\langle V \| e \rangle \mapsto_{+,0,-} c$ implies $c \in \perp\!\!\!\perp$.
- $v \in Sat(\mathcal{A})$, $E \in \mathcal{C}$: analogous to the previous case by duality.
- $v, e \in Sat(\mathcal{A})$: Let us consider the possible head reductions:
  - $\langle v \| e \rangle \mapsto_0 c$: in this case, both v and e are simple (co-)values, and neither can take a positively charged head reduction step. Therefore, for $v, e \in Sat(\mathcal{A})$ to hold, it must be that $\langle v \| e \rangle \in \perp\!\!\!\perp$, and so $c \in \perp\!\!\!\perp$ because $\perp\!\!\!\perp$ is closed under reduction.
  - $\langle v \| e \rangle \mapsto_+ c$: in this case, e must be a co-value, so for $v \in Sat(\mathcal{A})$ to hold, either $\langle v \| e \rangle \in \perp\!\!\!\perp$ already or there is a c' such that $\langle v \| e \rangle \mapsto_+ c'$. In the former case, $c \in \perp\!\!\!\perp$ because $\perp\!\!\!\perp$ is closed under reduction, and in the latter case, $c = c' \in \perp\!\!\!\perp$ because $\mapsto_+$ is deterministic.
  - $\langle v \| e \rangle \mapsto_- c$: analogous to the previous case by duality. ☐

Lemma 16. *$Sat(\mathcal{A})$ is forward closed.*

Proof.
- Suppose $v \in Sat(\mathcal{A})$ and $v \to v'$. Let $E \in \mathcal{A}$, so that $v \in Sat(\mathcal{A})$ implies that either $v \perp\!\!\!\perp E$ or $\langle v \| E \rangle \mapsto_+ c$.
  In the first case we know that $\langle v \| E \rangle \in \perp\!\!\!\perp$, so $\langle v \| E \rangle \to \langle v' \| E \rangle$ and $\langle v' \| E \rangle \in \perp\!\!\!\perp$ since $\perp\!\!\!\perp$ is forward closed.
  In the second case we know that $\langle v' \| E \rangle \leftarrow \langle v \| E \rangle \mapsto_+ c \in \perp\!\!\!\perp$. Based on the commutation of internal and head reduction (Lemma 10), and the fact that $\perp\!\!\!\perp$ is forward closed, we have one of two cases:
  - $c \twoheadrightarrow \langle v' \| E \rangle \in \perp\!\!\!\perp$, or
  - $\langle v' \| E \rangle \mapsto_+ c' \twoheadleftarrow c \in \perp\!\!\!\perp$

  Therefore, since in any case $\langle v' \| E \rangle \in \perp\!\!\!\perp$ or $\langle v' \| E \rangle \mapsto_+ c' \in \perp\!\!\!\perp$, then $v' \in Sat(\mathcal{A})$.
- Suppose $e \in Sat(\mathcal{A})$ and $e \to e'$. Analogous to the previous case by duality. ☐

Corollary 6. *Given $\mathcal{A} = Step_{\mathcal{C}}(\mathcal{A})$, $\mathcal{A}$ is forward closed when $\mathcal{C} \in TypeCore$.*

Proof. Any $v, e \in \mathcal{A}$ is in either $\mathcal{C}$ or $Sat(\mathcal{A})$, so it follows by Lemma 16 and definition of $TypeCore$. ☐

Corollary 7. *For any $\mathcal{C} \in TypeCore$, $\mathcal{R}(\mathcal{C})$ is a reducibility candidate such that $\mathcal{C} \sqsubseteq \mathcal{R}(\mathcal{C})$.*

Proof. By Lemma 5, $\mathcal{R}(\mathcal{C}) = Step_{\mathcal{C}}(\mathcal{R}(\mathcal{C}))$, so by definition we know that $\mathcal{C} \sqsubseteq \mathcal{R}(\mathcal{C})$. Furthermore, we know that both $Head(\mathcal{R}(C))$ and $\mathcal{R}(\mathcal{C})^{\perp}$ refine $Sat(\mathcal{R}(\mathcal{C}))$, which in turn refines $\mathcal{R}(\mathcal{C})$. Furthermore, by definition of $TypeCore$ and Sat, we know that $\mathcal{R}(\mathcal{C}) \sqsubseteq \mathcal{W}$. Finally, by Corollary 6 and Lemma 15 we know that $\mathcal{R}(\mathcal{C})$ is forward closed and all head reducts are in $\perp\!\!\!\perp$, so $\mathcal{R}(\mathcal{C}) \sqsubseteq \mathcal{R}(\mathcal{C})^{\perp}$ by Lemma 13. Therefore, $\mathcal{R}(\mathcal{C})$ is a reducibility candidate. ☐

E. The Model

We use the notation $\mathbb{O}$ to denote a set of ordinals that is sufficiently large for our purposes: $\mathbb{O}$ contains all ordinals less than or equal to $\omega \times 2$. However, any larger set of ordinals would serve just as well.

Definition 3. *PK, or* pre-kinds *is the smallest set such that*

1. *$\mathbb{O} \in PK$,*
2. *$CR \in PK$, and*
3. *$\forall a, b \in PK, \{f : a \rightharpoonup b\} \in PK$.*

The universe $\mathcal{U}$ is defined as the union of PK

$$\mathcal{U} = \bigcup_{a \in PK} a$$

Definition 4. *A* semantic kind *$\kappa \in SKind$ is a pair (S, R) where $S \in \mathcal{P}(\mathcal{U})$ and $R \in \mathcal{P}(Type \times S)$. We say that S is the* domain *of the semantic kind and R is the* syntactic-semantic relation*.*

In order to define the meaning of types as a generic fold parameterized by an interpretation operation, we define an intermediate stage between syntactic and semantic types called hybrid types.

Definition 5. *The set of* hybrid types *(HType) consists of*

1. *type variables,*
2. $\mathsf{F}(\overrightarrow{\mathcal{A}})$ *where* $\overrightarrow{\mathcal{A} \in \mathcal{U}}$ *and* $\mathsf{F} \in \mathcal{F}$*,*
3. *the constants* 0 *and* ∞*,*
4. $+1(\mathcal{M})$ *where* $\mathcal{M} \in \mathcal{U}$*.*

We define the interpretation functions on types and kinds using the same notation $[\![-]\!]$, which is disambiguated by context. Intuitively, this interpretation is parameterized by an operation, $\mathcal{H}$, which does the real hard work of assigning a semantic meaning to syntactic types. The purpose of abstracting out $\mathcal{H}$ is to break up the many recursions involved with interpreting recursive (co-)data types, so that the outermost interpretation is defined structurally over just the syntax of types, without concern on how the different types are related to one another.

$$\begin{aligned} [\![k \in Kind]\!] &: (HType \rightharpoonup \mathcal{U}) \rightharpoonup SKind \\ [\![A \in Type]\!] &: (HType \rightharpoonup \mathcal{U}) \rightharpoonup \mathcal{U} \end{aligned}$$

The interpretation of types and kinds is given by structural induction over the syntax. Technically, we consider the interpretation giving the two components of semantic kinds, the domain and syntactic-semantic relation, as two separate functions. The interpretation of types and and the domain of kinds are defined by mutual induction over the structure of syntactic types and kinds, and the interpretation of the syntactic-semantic relation of kinds is defined by structural induction on kinds, and using the previous definition.

$$\begin{aligned} [\![a]\!]_{\mathcal{H}} &\triangleq \mathcal{H}(a) \\ [\![0]\!]_{\mathcal{H}} &\triangleq \mathcal{H}(0) \\ [\![\infty]\!]_{\mathcal{H}} &\triangleq \mathcal{H}(\infty) \\ [\![N+1]\!]_{\mathcal{H}} &\triangleq \mathcal{H}(+1([\![N]\!]_{\mathcal{H}})) \\ [\![\mathsf{F}(\overrightarrow{A})]\!]_{\mathcal{H}} &\triangleq \mathcal{H}(\mathsf{F}(\overrightarrow{[\![A]\!]_{\mathcal{H}}})) \\ [\![\lambda a : k.B]\!]_{\mathcal{H}} &\triangleq \lambda \mathcal{A} \in \pi_1([\![k]\!]_{\mathcal{H}}).[\![B]\!]_{\mathcal{H}\{\mathcal{A}/a\}} \\ [\![A\ B]\!]_{\mathcal{H}} &\triangleq [\![A]\!]_{\mathcal{H}}([\![B]\!]_{\mathcal{H}}) \end{aligned}$$

$$\begin{aligned} \pi_1([\![\star]\!]_{\mathcal{H}}) &\triangleq CR \\ \pi_1([\![k_1 \to k_2]\!]_{\mathcal{H}}) &\triangleq \pi_1([\![k_2]\!]_{\mathcal{H}})^{\pi_1([\![k_1]\!]_{\mathcal{H}})} \\ \pi_1([\![\mathsf{Ix}]\!]_{\mathcal{H}}) &\triangleq \{\mathcal{N} \in \mathbb{O} | \exists N \in Type, N \sim_{\mathcal{H}} \mathcal{N}\} \\ \pi_1([\![\mathsf{Ord}]\!]_{\mathcal{H}}) &\triangleq \mathbb{O} \\ \pi_1([\![< N]\!]_{\mathcal{H}}) &\triangleq \{\mathcal{M} \in \mathbb{O} \\ &\qquad |\exists N' \in Type. N \twoheadrightarrow_\beta N' \wedge \mathcal{M} < [\![N']\!]_{\mathcal{H}}\} \end{aligned}$$

$$\begin{aligned} \pi_2([\![\star]\!]_{\mathcal{H}}) &\triangleq Type \times CR \\ \pi_2([\![k_1 \to k_2]\!]_{\mathcal{H}}) &\triangleq \{(A, \mathcal{A}) | \forall (B, \mathcal{B}) \in \pi_2([\![k_1]\!]_{\mathcal{H}}). \\ &\qquad (A\ B, \mathcal{A}(\mathcal{B})) \in \pi_2([\![k_2]\!]_{\mathcal{H}})\} \\ \pi_2([\![\mathsf{Ix}]\!]_{\mathcal{H}}) &\triangleq \{(M, \mathcal{M}) | M \sim_{\mathcal{H}} \mathcal{M}\} \\ \pi_2([\![\mathsf{Ord}]\!]_{\mathcal{H}}) &\triangleq \{(M, \mathcal{M}) | \exists M' \in Type. M \twoheadrightarrow_\beta M' \\ &\qquad \wedge [\![M']\!]_{\mathcal{H}} \leq \mathcal{M}\} \\ \pi_2([\![< N]\!]_{\mathcal{H}}) &\triangleq \{(M, \mathcal{M}) | \exists M' \in Type. \\ &\qquad M \twoheadrightarrow_\beta M' \\ &\qquad \wedge [\![M']\!]_{\mathcal{H}} \leq \mathcal{M} < [\![N]\!]_{\mathcal{H}}\} \end{aligned}$$

Note that the relation $\sim_{\mathcal{H}}$ used in $[\![\mathsf{Ix}]\!]_{\mathcal{H}}$ is defined as the smallest subset of Type $\times\ \mathbb{O}$ such that

1. $0 \sim_{\mathcal{H}} \mathcal{H}(0)$ when $\mathcal{H}(0) \in \mathbb{O}$,
2. for all $N \sim_{\mathcal{H}} \mathcal{N}$, $N+1 \sim_{\mathcal{H}} \mathcal{H}(+1(\mathcal{N}))$ when $\mathcal{H}(+1(\mathcal{N})) \in \mathbb{O}$.

The application $[\![A]\!]_{\mathcal{H}}([\![B]\!]_{\mathcal{H}})$ is defined whenever there exists $\kappa_1, \kappa_2 \in PK$ such that $[\![A]\!]_{\mathcal{H}} \in \kappa_1 \rightharpoonup \kappa_2$ and $[\![B]\!]_{\mathcal{H}} \in \kappa_1$, otherwise it is undefined. We will use the shorthand notation $\mathcal{A} \in [\![k]\!]_{\mathcal{H}}$ when k is a kind to indicate that $\mathcal{A}$ is an element of $\pi_1([\![k]\!]_{\mathcal{H}})$, and $A\ [\![k]\!]_{\mathcal{H}}\ \mathcal{A}$ when $(A, \mathcal{A})$ is an element of $\pi_2([\![k]\!]_{\mathcal{H}})$.

Also note that we give an interpretation of sorts, as well. The sort of non-erasable kinds, $\blacksquare$, is interpreted as the whole set of all semantic kinds, and the sort of erasable kinds, $\square$, adds the restriction that the syntactic-semantic relation is backwards closed under β reduction.

$$\begin{aligned} [\![\blacksquare]\!] &\triangleq SKind \\ [\![\square]\!] &\triangleq \{\kappa \in SKind \\ &\qquad |\forall (A, \mathcal{A}) \in \pi_2(\kappa). A' \twoheadrightarrow_\beta A \implies (A', \mathcal{A}) \in \pi_2(\kappa)\} \end{aligned}$$

The main work of defining types is in giving the core interpretation of type constructors in $\mathcal{F}$:

$$\langle\!\langle A \in HType \rangle\!\rangle : (HType \rightharpoonup \mathcal{U}) \rightharpoonup TypeCore$$

First, we specify when an interpretation operation $\mathcal{H}$ assigns a plausible meaning to the numeric measures used in types.

Definition 6. *A map* $\mathcal{H} : HType \rightharpoonup \mathcal{U}$ *is plausible if*

1. $\mathcal{H}(0) \in \mathbb{O}$*,*
2. $\mathcal{H}(\infty) \in \mathbb{O}$*,*
3. *for all* $\mathcal{N} \in \mathbb{O}$*,* $\mathcal{H}(+1(\mathcal{N})) \in \mathbb{O}$*,*
4. $\mathcal{H}(0) < \mathcal{H}(\infty)$*,*
5. *for all* $\mathcal{N} \in \mathbb{O}$*,* $\mathcal{N} < \mathcal{H}(\infty)$ *implies* $\mathcal{H}(+1(\mathcal{N})) < \mathcal{H}(\infty)$*, and*
6. *for all* $\mathcal{M}, \mathcal{N} \in \mathbb{O}$*,* $\mathcal{M} < \mathcal{N}$ *implies* $\mathcal{M} < \mathcal{H}(+1(\mathcal{M})) \leq \mathcal{N}$*.*

We assume that $\mathcal{F}$ comes with some ordering among hybrid types that is both well-founded but also enough that the core interpretation of types is well-defined with respect to $\langle\!\langle - \rangle\!\rangle$. Note that we say two maps $\mathcal{H}, \mathcal{I} : HType \rightharpoonup \mathcal{U}$ are equivalent up to a hybrid type h (written $\mathcal{H} \equiv_{<h} \mathcal{I}$) if and only if for all $g < h \in HType$, $\mathcal{H}(g) = \mathcal{I}(g)$ and are defined.

Definition 7. *The set of type constructors* $\mathcal{F}$ *is well-founded if there is a partial well order* $<$ *on the set of hybrid types formable from* $\mathcal{F}$ *such that for any plausible* $\mathcal{H}$*:*

1. *Given any* $\mathsf{F}(\overrightarrow{a : k_1}) : k_2 \in \mathcal{F}$ *and* $\overrightarrow{\mathcal{A} \in [\![k_1]\!]_{\mathcal{H}}}$*,* $\langle\!\langle \mathsf{F}(\overrightarrow{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$ *is defined and in* $[\![k_2]\!]_{\mathcal{H}}$ *whenever for all* $\mathsf{G}(\overrightarrow{b : k_3}) : k_4 \in \mathcal{F}$ *and* $\overrightarrow{\mathcal{B} \in [\![k_3]\!]_{\mathcal{H}}}$ *such that* $\mathsf{G}(\overrightarrow{\mathcal{B}}) < \mathsf{F}(\overrightarrow{\mathcal{A}})$*,* $\mathcal{H}(\mathsf{G}(\overrightarrow{\mathcal{B}}))$ *is defined and in* $[\![k_4]\!]_{\mathcal{H}}$*.*
2. *For any other plausible* $\mathcal{I}$ *and for any* $h \in HType$ *such that* $\mathcal{H} \equiv_{<h} \mathcal{I}$*,* $\langle\!\langle h \rangle\!\rangle_{\mathcal{H}} = \langle\!\langle h \rangle\!\rangle_{\mathcal{I}}$*.*

Definition 8. *The set* $[\![\mathcal{F}]\!]$ *consists of partial functions* $HType \rightharpoonup \mathcal{U}$ *such that if* $\mathcal{H} \in [\![\mathcal{F}]\!]$ *then*

1. *for any* $\mathsf{F}(\overrightarrow{a : k}) : k' \in \mathcal{F}$ *and any* $\overrightarrow{\mathcal{A} \in [\![k]\!]_{\mathcal{H}}}$*,* $\langle\!\langle \mathsf{F}(\overrightarrow{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}} \sqsubseteq \mathcal{H}(\mathsf{F}(\overrightarrow{\mathcal{A}})) \in [\![k']\!]_{\mathcal{H}}$*, and*
2. $\mathcal{H}$ *is plausible.*

Lemma 17. *Given any well-founded set of type constructors* $\mathcal{F}$ *and* $\mathcal{I} : HType \rightharpoonup \mathcal{U}$ *there exists a* $\mathcal{H} \in [\![\mathcal{F}]\!]$ *such that for any* $h = 0, \infty, +1(\mathcal{M}), a$ *we have* $\mathcal{H}(h) = \mathcal{I}(h)$*.*

Proof. We define the function:

$$next : (HType \rightharpoonup \mathcal{U}) \to HType \rightharpoonup \mathcal{U}$$

$$next(\mathcal{H})(h) = \begin{cases} \mathcal{R}(\langle\!\langle \mathsf{F}(\vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}) & \text{if } h = \mathsf{F}(\vec{\mathcal{A}}) \text{ and } \mathsf{F} \in \mathcal{F} \\ \mathcal{I}(h) & \text{otherwise} \end{cases}$$

The conditions we require for well-foundedness of $\mathcal{F}$ are exactly what we need to take a fixed point of this function and show that it is in $[\![\mathcal{F}]\!]$.

Now, by the second property of Definition 7 above, we know that for all h and $\mathcal{H} \equiv_{<h} \mathcal{H}'$, $next(\mathcal{H}) \equiv_{<h} next(\mathcal{H}')$ and so $next(\mathcal{H})(h) = next(\mathcal{H}')(h)$ as well. Using this fact, we show that $next$ has a (unique) fixed point. That is, we wish to find the $\mathcal{H}_\Box$ such that $next(\mathcal{H}_\Box) = \mathcal{H}_\Box$. The proof proceeds by induction on h by the well-founded order given for $\mathcal{F}$. For the inductive hypothesis, we get that there exists a map $\mathcal{H}'_\Box$ such that for all $g < h$, $\mathcal{H}'_\Box(g) = next(\mathcal{H}'_\Box)(g)$. In other words, $\mathcal{H}'_\Box \equiv_{<h} next(\mathcal{H}'_\Box)$ and so $next(\mathcal{H}'_\Box)(h) = next(next(\mathcal{H}'_\Box))(h)$. Thus, $\mathcal{H}_\Box = next(\mathcal{H}'_\Box)$ and $\mathcal{H}_\Box(h)$ is the unique possible value for h. That is, we know that any other $\mathcal{H}'' \equiv_{<h} \mathcal{H}_\Box$ such that $\mathcal{H}''(h) = next(\mathcal{H}'_\Box)(h)$ has the property that $\mathcal{H}'_\Box \equiv_{<h} \mathcal{H}'' \equiv_{<h} next(\mathcal{H}'_\Box) \equiv_{<h} \mathcal{H}_\Box$ and $next(\mathcal{H}'')(h) = next(\mathcal{H}_\Box)(h) = \mathcal{H}_\Box(h)$.

Thus, there is a unique $\mathcal{H}_\Box$ such that $next(\mathcal{H}_\Box) = \mathcal{H}_\Box$. We need only show that this $\mathcal{H}_\Box \in [\![\mathcal{F}]\!]$. That, is for any $\mathsf{F}(\overrightarrow{a : k}) : k \in \mathcal{F}$ and $\overrightarrow{\mathcal{A} \in [\![k]\!]_{\mathcal{H}_\Box}}$, $\langle\!\langle \mathsf{F}(\vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}_\Box} \sqsubseteq \mathcal{H}_\Box(\mathsf{F}(\vec{\mathcal{A}})) \in [\![k']\!]_{\mathcal{H}_\Box}$. Again, we proceed by well founded induction on the set of hybrid types, and what we need to show for the inductive step is exactly the definition of $next$, the fact that $\mathcal{C} \sqsubseteq \mathcal{R}(\mathcal{C})$, and the first property of Definition 7 above. □

We interpret the kinding environment $[\![\Theta]\!]_{\mathcal{H}}$ as a pair of $S \subset HType \rightharpoonup \mathcal{U}$ and a relation on substitutions and S

$$\begin{aligned} [\![-]\!]_{\mathcal{H}} &= (\mathcal{H}, \{(\theta, \mathcal{H})\}) \\ [\![\Theta, a : k]\!]_{\mathcal{H}} &= (\{\mathcal{I} \mid \exists \mathcal{I}' \in \pi_1([\![\Theta]\!]_{\mathcal{H}}). \\ &\qquad \mathcal{I}(a) \in [\![k]\!]_{\mathcal{I}'} \\ &\qquad \wedge \forall h \in HType, h \neq a \Rightarrow \mathcal{I}(h) = \mathcal{I}'(h)\}. \\ &\quad \{(\theta, \mathcal{I}) \mid \exists (\theta', \mathcal{I}') \in \pi_2([\![\Theta]\!]_{\mathcal{H}}). \\ &\qquad \theta(a)\ [\![k]\!]_{\mathcal{I}}\ \mathcal{I} \wedge (\forall b \neq a. \theta(b) = \theta'(b)) \\ &\qquad \wedge (\forall h \in HType. h \neq a \Rightarrow \mathcal{I}(h) = \mathcal{I}'(h))\} \end{aligned}$$

We use the notation $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ if $\mathcal{I} \in \pi_1([\![\Theta]\!]_{\mathcal{H}})$ and $\theta [\![\Theta]\!]_{\mathcal{H}} \mathcal{I}$ if $(\theta, \mathcal{I}) \in \pi_2([\![\Theta]\!]_{\mathcal{H}})$.

Lemma 18. *For any $\mathcal{I}$ in $[\![\Theta]\!]_{\mathcal{H}}$, $\mathcal{I}(h) = \mathcal{H}(h)$ whenever h is not a variable.*

Proof. By induction on Θ. □

Corollary 8. *For any $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$, if $\mathcal{H} \in [\![\mathcal{F}]\!]$ then $\mathcal{I} \in [\![\mathcal{F}]\!]$.*

Lemma 19. *For any Θ, if $[\![\Theta]\!]_{\mathcal{H}}$ is defined and for every $a : k \in \Theta$ we know $k \neq \overrightarrow{k \to} < 0 \wedge k \neq \overrightarrow{k \to} < i$, then there exist θ and $\mathcal{I}$ such that $\theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}$.*

Proof. All kinds except $\overrightarrow{k \to} < 0$ and $\overrightarrow{k \to} < i$ are inhabited. The kind $\star$ is inhabited by $(a, \mathcal{R}((\emptyset, \emptyset)))$, the kinds Ix, Ord and $< \infty$ are inhabited by $(0, \mathcal{H}(0))$, and $< N + 1$ is inhabited by $(N, [\![N]\!]_{\mathcal{H}})$. Function kinds are inhabited whenever their final result kind is, since we can always build the constant function. □

Lemma 20. *If θ is a substitution which only replaces type variables, and $c\{\theta\} \in \perp\!\!\!\perp$ then $c \in \perp\!\!\!\perp$.*

Proof. This follows from the fact that if $c \to c'$, then $c\{\theta\} \to c'\{\theta\}$, so strong normalization of $c\{\theta\}$ implies strong normalization of c. The main point is that filling in types for type variables can only allow for *more* reductions, either due to filling in the index of an $\mathsf{Inflate}$ or $\mathsf{Deflate}$ structure or by specifying that the upper bound for a quantified type variable in a case abstraction is inhabited. □

By contrast, we interpret $[\![\Gamma]\!]_{\mathcal{H}}$ and $[\![\Delta]\!]_{\mathcal{H}}$ as just sets of substitutions.

$$\begin{aligned} \gamma \in [\![\Gamma]\!]_{\mathcal{H}} &\iff \forall x : A \in \Gamma. x\{\gamma\} \in Val([\![A]\!]_{\mathcal{H}}) \\ \delta \in [\![\Delta]\!]_{\mathcal{H}} &\iff \forall \alpha : A \in \Delta. \alpha\{\delta\} \in Val([\![A]\!]_{\mathcal{H}}) \end{aligned}$$

Note that Val is the function on pre-types that keeps only their (co-)values.

Lemma 21. *1. If $[\![\Gamma]\!]_{\mathcal{H}}$ is defined then it is inhabited by the identity substitution.*
2. If $[\![\Delta]\!]_{\mathcal{H}}$ is defined then it is inhabited by the identity substitution.

Proof. If $[\![\Gamma]\!]_{\mathcal{H}}$ is defined then $\forall (x : A) \in \Gamma$, $[\![A]\!]_{\mathcal{H}}$ is defined and must yield a reducibility candidate since $Val([\![A]\!]_{\mathcal{H}})$ is defined. Since variables inhabit every reducibility candidate (Lemma 2), and are (co-)values (because we assume the chosen strategy is focalizing), the identity substitution inhabits $[\![\Gamma]\!]_{\mathcal{H}}$ (and by duality $[\![\Delta]\!]_{\mathcal{H}}$). □

Note that we denote the extension of an interpretation operation $\mathcal{H}$ with the interpretation $\mathcal{B}$ for a type variable b as $\mathcal{H}\{\mathcal{B}/b\}$, where

$$\mathcal{H}\{\mathcal{B}/b\}(h) \triangleq \begin{cases} \mathcal{B} & \text{if } h = b \\ \mathcal{H}(h) & \text{otherwise} \end{cases}$$

Lemma 22. *If $[\![B]\!]_{\mathcal{H}}$ is defined and $[\![A]\!]_{\mathcal{H}\{[\![B]\!]_{\mathcal{H}}/b\}}$ is defined, then $[\![A]\!]_{\mathcal{H}\{[\![B]\!]_{\mathcal{H}}/b\}} = [\![A\{B/b\}]\!]_{\mathcal{H}}$.*

Proof. By induction on the structure of A. □

F. Interpreting Specific Types

To define the core meaning following an orthogonality-based methodology, except that instead of attempting to build a reducibility candidate through the double-orthogonal closure, we build a simplified interpretation in $TypeCore$ using the $\perp s$ negation operation inside $Simp(\mathcal{W})$.

On the one hand, for the hybrid types of data type constructors, h, we build this core definition through some set of focalized constructions, $Cons_{\mathcal{H}}(h)$, from the $\perp s \perp s$ closure such that

$$\begin{aligned} v \in \langle\!\langle h \rangle\!\rangle_{\mathcal{H}} &\iff v \in (Cons_{\mathcal{H}}(h), \emptyset)^{\perp s \perp s} \\ e \in \langle\!\langle h \rangle\!\rangle_{\mathcal{H}} &\iff e \in (Cons_{\mathcal{H}}(h), \emptyset)^{\perp s} \end{aligned}$$

On the other hand, for the hybrid types of co-data type constructors, g, we build this core definition through some set of focalized observations, $Obs_{\mathcal{H}}(h)$, from the $\perp s \perp s$ closure such that

$$\begin{aligned} v \in \langle\!\langle g \rangle\!\rangle_{\mathcal{H}} &\iff v \in (\emptyset, Obs_{\mathcal{H}}(g))^{\perp s} \\ e \in \langle\!\langle g \rangle\!\rangle_{\mathcal{H}} &\iff e \in (\emptyset, Obs_{\mathcal{H}}(g))^{\perp s \perp s} \end{aligned}$$

Note that in both cases, $\langle\!\langle h \rangle\!\rangle_{\mathcal{H}} = \langle\!\langle h \rangle\!\rangle_{\mathcal{H}}{}^{\perp s}$. It follows that this definition is always in $TypeCore$. Furthermore by double negation introduction (Lemma 7), in the case of hybrid types for data type constructors h, we are ensured that $(Cons_{\mathcal{H}}(h), \emptyset) \sqsubseteq \langle\!\langle h \rangle\!\rangle_{\mathcal{H}}$ whenever $Cons_{\mathcal{H}}(h)$ is a set of strongly-normalizing simple values, and in the case of hybrid types for co-data type constructors g, $(\emptyset, Obs_{\mathcal{H}}(g)) \sqsubseteq \langle\!\langle g \rangle\!\rangle_{\mathcal{H}}$ whenever $Obs_{\mathcal{H}}(g)$ is a set of strongly-normalizing simple co-values.

To show that our set of type constructors, $\mathcal{F}$, is well-founded according to Definition 7, we suppose for simplicity that the declarations are given in a linear manner, so that declarations can only refer to the type constructors introduced by previous declarations, which is ensured by their checks for well-formedness. If a (co-)data type declaration refers to a type constructor that hasn't yet been declared, then the sequents for its constructors can't be well-formed by the rules we have available. This gives us the main basis for the well-founded ordering for $\mathcal{F}$: type constructors can only be greater than other type constructors earlier in the line. The rest of the relations among hybrid types of $\mathcal{F}$ are given by the possible recursive nature of the declaration itself.

F.1 Non-recursive (Co-)data Declarations

For any declared data type of the form

$$\mathbf{data}\,\mathsf{F}(\overrightarrow{a:k})\;\mathbf{where}$$
$$\overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{a}) | \vec{C}}$$

We require that for all $\mathsf{G}(\overrightarrow{d:k'}) : \star \in \mathcal{F}$, $\mathsf{G}(\vec{\mathcal{D}}) < \mathsf{F}(\mathcal{N}, \vec{\mathcal{A}})$. This allows us to give a $Cons$truction-oriented definition for $\langle\!\langle \mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$ such that the order of hybrid types satisfies Definition 7. In particular, $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}))$ is defined as:

$$
\begin{aligned}
&Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))\\
&\triangleq \{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x})\{\gamma, \delta, \theta\} |\\
&\qquad \exists\, \mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{a}) | \vec{C} \in \overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{a}) | \vec{C}}.\\
&\qquad \exists \theta\, [\![\overrightarrow{d:k'}]\!]_{\mathcal{H}\{\overrightarrow{\mathcal{A}/a}\}}\, \mathcal{I}. \exists \gamma \in [\![\overrightarrow{x:B}]\!]_{\mathcal{I}}, \delta \in [\![\overrightarrow{\alpha:C}]\!]_{\mathcal{I}}\}.
\end{aligned}
$$

Lemma 23. *If $\mathcal{F}$ is well-founded, $\mathcal{H} \in [\![\mathcal{F}]\!]$ and the declaration of* F *is well-formed with respect to $\mathcal{F}$, then $Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$ is a defined pre-type for all $\overrightarrow{\mathcal{A} \in [\![k]\!]_{\mathcal{H}}}$.*

Proof. Because the declaration of F is well-formed, we know that the sequent for each of its constructors, $\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{a}) | \vec{C}$, is also well-formed, $(\overrightarrow{x:B} \vdash_{\overrightarrow{a:k}, \overrightarrow{d:k'}} \overrightarrow{\alpha:C})\,\mathbf{seq}$. By the soundness of kinding rules (shown later in Theorem 35), we therefore know that each sequent is soundly represented in the model, $[\![(\overrightarrow{x:B} \vdash_{\overrightarrow{a:k}, \overrightarrow{d:k'}} \overrightarrow{\alpha:C})\,\mathbf{seq}\,]\!]_{\mathcal{H}}$. Therefore, $Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$ is a defined pre-type. □

As a corollary, we have that if $Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$ is defined then $\langle\!\langle \mathsf{F}(\vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$ is in $TypeCore$.

Lemma 24. *If $Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$ is defined, then it is a set of simple values in $\mathcal{W}$.*

Proof. Given any construction $v = \mathsf{K}^{\overrightarrow{d:k}}(\vec{\alpha}, \vec{x})\{\theta, \gamma, \delta\} \in Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$, we know that the substitutions γ and δ only ever replace the (co-)variables $\vec{\alpha}$ and $\vec{x}$ with strongly-normalizing (co-)values, since the substitutions only range over the (co-)values in reducibility candidates when defined (and all reducibility candidates refine $\mathcal{W}$). This means that v is simple, since it cannot take part in a positive head reduction (lifting never applies), and is therefore also a value because we assume our chosen strategy is focalizing. Furthermore, $v \in \mathcal{W}$ because all of its sub-(co-)-terms are strongly normalizing, and no other reductions apply to v. □

As a corollary, we have that if $Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$ is defined and contains a term v, then $v \in \langle\!\langle \mathsf{F}(\vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$.

Lemma 25. *If $Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$ is defined and E is the case abstraction $\tilde{\mu}[\overrightarrow{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c}]$ such that*

$$
\begin{aligned}
&\forall\, \mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{a}) | \vec{C} \in \overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{a}) | \vec{C}}.\\
&\exists \mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c \in \overrightarrow{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c}.\\
&\forall \theta\, [\![\overrightarrow{d:k'}]\!]_{\mathcal{H}\{\overrightarrow{\mathcal{A}/a}\}}\, \mathcal{I}. \forall \gamma \in [\![\overrightarrow{x:B}]\!]_{\mathcal{I}}, \delta \in [\![\overrightarrow{\alpha:C}]\!]_{\mathcal{I}}.\\
&\quad c\{\theta, \gamma, \delta\} \in \perp\!\!\!\perp
\end{aligned}
$$

then $E \in \langle\!\langle \mathsf{F}(\vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$.

Proof. We must show that $E \in \mathcal{W}$ and $\langle V \| E \rangle \in \perp\!\!\!\perp$ for every $V \in Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$.

- $E \in \mathcal{W}$: for each sub-command c in E we have two cases, depending on whether or not any kind k' of the quantified type variables $\overrightarrow{d:k'}$ introduced by the pattern has the form $\overrightarrow{k'' \to} < 0$ or $\overrightarrow{k'' \to} < i$:
  - There is a k' such that $\overrightarrow{k'' \to} < 0$ or $\overrightarrow{k'' \to} < i$: then by the caveat on reduction inside a case abstraction, $c \not\mapsto$, so $c \in \perp\!\!\!\perp$.
  - There is no k' such that $\overrightarrow{k'' \to} < 0$ or $\overrightarrow{k'' \to} < i$: then by Lemma 19, there is a $\theta\, [\![\overrightarrow{d:k'}]\!]_{\mathcal{H}\{\overrightarrow{\mathcal{A}/a}\}}\, \mathcal{I}$, and by Lemma 21, the identity substitutions inhabit $[\![\overrightarrow{x:B}]\!]_{\mathcal{I}}$ and $[\![\overrightarrow{\alpha:C}]\!]_{\mathcal{I}}$. Therefore, we know that $c\{\theta\} \in \perp\!\!\!\perp$, and so $c \in \perp\!\!\!\perp$ by Lemma 20.

  In either case, all sub-commands of E are in $\perp\!\!\!\perp$, so $E \in \mathcal{W}$.
- $\langle V \| E \rangle \in \perp\!\!\!\perp$ for every $V \in Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$: note that for every such command, $\langle V \| E \rangle \mapsto_0 c$ and $c \in \perp\!\!\!\perp$ by definition of $Cons_{\mathcal{H}}(\mathsf{F}(\vec{\mathcal{A}}))$. Therefore, $\langle V \| E \rangle \in \perp\!\!\!\perp$ by Lemma 14. □

The case for non-recursive co-data type declarations follows analogously by duality.

F.2 (Co-)data Declarations by Noetherian Recursion

Declarations by noetherian recursion is largely analogous to the non-recursive declarations shown previously, and we highlight the difference here. For any declared data type of the form:

$$\mathbf{data}\,\mathsf{F}(i : \mathrm{Ord}, \overrightarrow{a:k})\;\mathbf{by}\text{ noetherian recursion on } i\;\mathbf{where}$$
$$\overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(i, \vec{a}) | \vec{C}}$$

In addition to the normal ordering of hybrid types for non-recursive declarations, we also have $\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}) < \mathsf{F}(\mathcal{M}, \vec{\mathcal{B}})$ whenever $\mathcal{N} < \mathcal{M} \in \mathbb{O}$. This allows us to give a $Cons$truction-oriented definition for $\langle\!\langle \mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$ such that F can refer to itself for smaller choices of the ordinal index. In particular, $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}))$ is defined as:

$$
\begin{aligned}
&Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}))\\
&\triangleq \{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x})\{\gamma, \delta, \theta\} |\\
&\qquad \exists\, \mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(i, \vec{a}) | \vec{C} \in \overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(i, \vec{a}) | \vec{C}}.\\
&\qquad \exists \theta\, [\![\overrightarrow{d:k'}]\!]_{\mathcal{H}'}\, \mathcal{I}. \exists \gamma \in [\![\overrightarrow{x:B}]\!]_{\mathcal{I}}, \delta \in [\![\overrightarrow{\alpha:C}]\!]_{\mathcal{I}}\}.\\
&\mathbf{where}\,\mathcal{H}' = \mathcal{H}\{\mathcal{N}/i, \overrightarrow{\mathcal{A}/a}\}
\end{aligned}
$$

Lemma 26. *If $\mathcal{F}$ is well-founded, $\mathcal{H} \in [\![\mathcal{F}]\!]$ and the declaration of* F *is well-formed with respect to $\mathcal{F}$, then $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}))$ is a defined pre-type for all $\mathcal{N} \in \mathbb{O}$ and $\overrightarrow{\mathcal{A} \in [\![k]\!]_{\mathcal{H}}}$.*

Proof. Because the well-formedness check for F can assume that F is already well-formed at smaller indices:

$$\frac{\Theta, i : \mathsf{Ord}, \Theta' \vdash M < i \qquad \overrightarrow{\Theta, i : \mathsf{Ord}, \Theta' \vdash A : k}}{\Theta, i : \mathsf{Ord}, \Theta' \vdash \mathsf{F}(M, \vec{A}) : \star}$$

we need to proceed by noetherian induction on the ordinal $\mathcal{N}$. Besides this difference, the proof follows analogously to Lemma 23. □

Lemma 27. *If $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}))$ is defined, then it is a set of simple values in $\mathcal{W}$.*

Proof. Analogous to the proof for Lemma 24. □

Lemma 28. *If $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}))$ is defined and E is the case abstraction $\tilde{\mu}[\overrightarrow{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c}]$ such that*

$$\begin{aligned}
&\forall\, \mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(i, \vec{a}) | \vec{C} \in \overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(i, \vec{a}) | \vec{C}}.\\
&\exists \mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c \in \overrightarrow{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c}.\\
&\forall \theta\, \overrightarrow{[\![d : k']\!]}_{\mathcal{H}\{\mathcal{N}/i, \overrightarrow{\mathcal{A}/a}\}}\, \mathcal{I}. \forall \gamma \in \overrightarrow{[\![x : B]\!]}_{\mathcal{I}}, \delta \in \overrightarrow{[\![\alpha : C]\!]}_{\mathcal{I}}.\\
&\quad c\{\theta, \gamma, \delta\} \in \perp\!\!\!\perp
\end{aligned}$$

then $E \in \langle\!\langle \mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$.

Proof. Analogous to the proof for Lemma 25. □

The case for a co-data type defined by noetherian recursion follows analogously by duality.

F.3 (Co-)data Declarations by Primitive Recursion

The constructions for a (co-)data type defined by primitive recursion are themselves defined by primitive recursion on the specified index: if the index is $\mathcal{H}(0)$ then the set of constructors used are those for for the 0 case; if the index equals $\mathcal{H}(+1(\mathcal{N}))$ for some $\mathcal{N} \in [\![Ix]\!]_{\mathcal{H}}$ then the successor case is chosen. For any declared data type of the form

$$\begin{aligned}
&\mathbf{data}\, \mathsf{F}(i : \mathsf{Ix}, \overrightarrow{a : k})\, \mathbf{by} \text{ by primitive recursion on } i\\
&\mathbf{where}\, i = 0 \qquad \overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(0, \vec{a}) | \vec{C}}\\
&\mathbf{where}\, i = j + 1 \quad \overrightarrow{\mathsf{K}' : \vec{B'} \vdash_{\overrightarrow{d':k''}} \mathsf{F}(0, \vec{a}) | \vec{C'}}
\end{aligned}$$

In addition to the normal ordering of hybrid types for non-recursive declarations, we also have the additional orderings: $\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}) < \mathsf{F}(\mathcal{H}(+1(\mathcal{N})), \vec{\mathcal{B}})$ for all $\mathcal{N} \in [\![\mathsf{Ix}]\!]_{\mathcal{H}}$. We now define the $Cons$tructions for F by primitive recursion on the index $\mathcal{N}$:

$$\begin{aligned}
&Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{H}(0), \vec{\mathcal{A}}))\\
&\triangleq \{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x})\{\gamma, \delta, \theta\} |\\
&\qquad \exists\, \mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(0, \vec{a}) | \vec{C} \in \overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(0, \vec{a}) | \vec{C}}.\\
&\qquad \exists \theta\, \overrightarrow{[\![d : k']\!]}_{\mathcal{H}'}\, \mathcal{I}. \exists \gamma \in \overrightarrow{[\![x : B]\!]}_{\mathcal{I}}, \delta \in \overrightarrow{[\![\alpha : C]\!]}_{\mathcal{I}}\}.\\
&\ \mathbf{where}\, \mathcal{H}' = \mathcal{H}\{\overrightarrow{\mathcal{A}/a}\}\\
&Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{H}(+1(\mathcal{N})), \vec{\mathcal{A}}))\\
&\triangleq \{\mathsf{K}'^{\overrightarrow{d':k''}}(\vec{\alpha}, \vec{x})\{\gamma, \delta, \theta\} |\\
&\qquad \exists\, \mathsf{K}' : \vec{B'} \vdash_{\overrightarrow{d':k''}} \mathsf{F}(j + 1, \vec{a}) | \vec{C'} \in\\
&\qquad\quad \overrightarrow{\mathsf{K}' : \vec{B'} \vdash_{\overrightarrow{d':k''}} \mathsf{F}(j + 1, \vec{a}) | \vec{C'}}.\\
&\qquad \exists \theta\, \overrightarrow{[\![d' : k'']\!]}_{\mathcal{H}'}\, \mathcal{I}. \exists \gamma \in \overrightarrow{[\![x : B']\!]}_{\mathcal{I}}, \delta \in \overrightarrow{[\![\alpha : C']\!]}_{\mathcal{I}}\}.\\
&\ \mathbf{where}\, \mathcal{H}' = \mathcal{H}\{\mathcal{N}/j, \overrightarrow{\mathcal{A}/a}\}
\end{aligned}$$

Lemma 29. *If $\mathcal{F}$ is well-founded, $\mathcal{H} \in [\![\mathcal{F}]\!]$ and the declaration of F is well-formed with respect to $\mathcal{F}$, then $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}))$ is a defined pre-type for all $\mathcal{N} \in \mathbb{O}$ and $\overrightarrow{\mathcal{A} \in [\![k]\!]_{\mathcal{H}}}$.*

Proof. Because the well-formedness checks for the successor constructors of F can assume that F is already well-formed the previous index j:

$$\frac{\overrightarrow{\Theta, j : \mathsf{Ix}, \Theta' \vdash A : k}}{\Theta, j : \mathsf{Ix}, \Theta' \vdash \mathsf{F}(j, \vec{A}) : \star}$$

we need to proceed by primitive induction on the ordinal $\mathcal{N}$. Besides this difference, the proof follows analogously to Lemma 23. □

Lemma 30. *If $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{N}, \vec{\mathcal{A}}))$ is defined, then it is a set of simple values in $\mathcal{W}$.*

Proof. Analogous to the proof for Lemma 24. □

Lemma 31. *1. If $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{H}(0), \vec{\mathcal{A}}))$ is defined and E is the case abstraction $\tilde{\mu}[\overrightarrow{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c}]$ such that*

$$\begin{aligned}
&\forall\, \mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(0, \vec{a}) | \vec{C} \in \overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(0, \vec{a}) | \vec{C}}.\\
&\exists \mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c \in \overrightarrow{\mathsf{K}^{\overrightarrow{d:k'}}(\vec{\alpha}, \vec{x}).c}.\\
&\forall \theta\, \overrightarrow{[\![d : k']\!]}_{\mathcal{H}\{\overrightarrow{\mathcal{A}/a}\}}\, \mathcal{I}. \forall \gamma \in \overrightarrow{[\![x : B]\!]}_{\mathcal{I}}, \delta \in \overrightarrow{[\![\alpha : C]\!]}_{\mathcal{I}}.\\
&\quad c\{\theta, \gamma, \delta\} \in \perp\!\!\!\perp
\end{aligned}$$

then $E \in \langle\!\langle \mathsf{F}(\mathcal{H}(0), \vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$.

2. If $Cons_{\mathcal{H}}(\mathsf{F}(\mathcal{H}(\mathcal{N}), \vec{\mathcal{A}}))$ is defined and E is the case abstraction $\tilde{\mu}[\overrightarrow{\mathsf{K}'^{\overrightarrow{d':k''}}(\vec{\alpha}, \vec{x}).c}]$ such that

$$\begin{aligned}
&\forall\, \mathsf{K}' : \vec{B'} \vdash_{\overrightarrow{d':k''}} \mathsf{F}(j + 1, \vec{a}) | \vec{C'}\\
&\qquad \in \overrightarrow{\mathsf{K}' : \vec{B'} \vdash_{\overrightarrow{d':k''}} \mathsf{F}(j + 1, \vec{a}) | \vec{C'}}.\\
&\exists \mathsf{K}'^{\overrightarrow{d':k''}}(\vec{\alpha}, \vec{x}).c \in \overrightarrow{\mathsf{K}'^{\overrightarrow{d':k''}}(\vec{\alpha}, \vec{x}).c}.\\
&\forall \theta\, \overrightarrow{[\![d' : k'']\!]}_{\mathcal{H}\{\mathcal{N}/j, \overrightarrow{\mathcal{A}/a}\}}\, \mathcal{I}. \forall \gamma \in \overrightarrow{[\![x : B']\!]}_{\mathcal{I}}, \delta \in \overrightarrow{[\![\alpha : C']\!]}_{\mathcal{I}}.\\
&\quad c\{\theta, \gamma, \delta\} \in \perp\!\!\!\perp
\end{aligned}$$

then $E \in \langle\!\langle \mathsf{F}(\mathcal{H}(\mathcal{N}), \vec{\mathcal{A}}) \rangle\!\rangle_{\mathcal{H}}$.

Proof. Analogous to the proof for Lemma 25 for both the zero and successor cases. □

F.4 Ascend and Descend

Interestingly, we do not need to include Ascend and Descend as part of the ordering relation on hybrid types. Taking the view that Descend and Ascend are just user defined types, we can compute their definitions as special cases of the general pattern:

$$\begin{aligned}&Cons_{\mathcal{H}}(\mathsf{Descend}(\mathcal{N},\mathcal{A}))\\&\triangleq \{\mathsf{Fall}^M(V) | M \llbracket < N \rrbracket_{\mathcal{H}}\ \mathcal{M}, V \in \mathcal{A}(\mathcal{N})\}\end{aligned}$$

By Lemma 26, we know that for all well-founded $\mathcal{F}$ and $\mathcal{H} \in \llbracket \mathcal{F} \rrbracket$, then $Cons_{\mathcal{H}}(\mathsf{Descend}(\mathcal{N},\mathcal{A}))$ is defined and $\langle\!\langle \mathsf{Descend}(\mathcal{N},\mathcal{A}) \rangle\!\rangle_{\mathcal{H}}$ is in $TypeCore$ for any $\mathcal{N} \in \mathbb{O}$ and $\mathcal{A} \in \llbracket \mathsf{Ord} \to \star \rrbracket_{\mathcal{H}}$. By Lemma 27, we know that any defined $Cons_{\mathcal{H}}(\mathsf{Descend}(\mathcal{N},\mathcal{A}))$ is included in $\langle\!\langle \mathsf{Descend}(\mathcal{N},\mathcal{A}) \rangle\!\rangle_{\mathcal{H}}$. And by Lemma 28 we know that $E = \tilde{\mu}[\mathsf{Fall}^{j<N}(x).c] \in \langle\!\langle \mathsf{Descend}(\mathcal{N},\mathcal{A}) \rangle\!\rangle_{\mathcal{H}}$ whenever

$$\forall M \llbracket < N \rrbracket_{\mathcal{H}} \mathcal{M}, V \in \mathcal{A}(\mathcal{M}).c\{V/x, M/j\} \in \perp\!\!\!\perp$$

For the special recursive form of case abstraction for Descend, we show that it is included in any family of reducibility candidates indexed by $\mathbb{O}$ which all include the non-recursive form.

Lemma 32 (Folding)**.** *Suppose that $\mathcal{A},\mathcal{B} : \mathbb{O} \to CR$ and $N \llbracket Ord \rrbracket_{\mathcal{H}}\ \mathcal{N}$ such that*

$$\forall M \llbracket < N \rrbracket_{\mathcal{H}}\ \mathcal{M}, E \in \mathcal{B}(\mathcal{M}).c\{M/j, E/\alpha\} \in \perp\!\!\!\perp$$

implies that

$$\tilde{\mu}[\mathsf{Fall}^{j<N}[\alpha].c] \in \mathcal{A}(\mathcal{N})$$

Then

$$\forall M \llbracket < N \rrbracket_{\mathcal{H}}\ \mathcal{M}, E \in \mathcal{B}(\mathcal{M}), v \in \mathcal{A}(\mathcal{M}).c\{M/j, E/\alpha, V/x\} \in \perp\!\!\!\perp$$

implies that

$$\tilde{\mu}[\mathsf{Fall}^{j<N}[\alpha](x).c] \in \mathcal{A}(\mathcal{N})$$

Proof. By noetherian induction on $\mathcal{N} \in \mathbb{O}$.

Note that the inductive hypothesis is that given any $\mathcal{M} < \mathcal{N}$ and $M\ \mathcal{O}\nabla\lceil[\mathcal{H}]\ \mathcal{M}$,

$$\forall O \llbracket < M \rrbracket_{\mathcal{H}}\ \mathcal{O}, E \in \mathcal{B}(\mathcal{O}), V \in \mathcal{A}(\mathcal{O}).c\{O/j, E/\alpha, V/x\} \in \perp\!\!\!\perp$$

implies that

$$\tilde{\mu}[\mathsf{Fall}^{j<M}[\alpha](x).c] \in \mathcal{A}(\mathcal{M})$$

We will now use the fact that $\mathcal{A}(\mathcal{N}) \in CR$ to prove that it contains $\tilde{\mu}[\mathsf{Fall}^{j<N}[\alpha](x).c] \in \mathcal{A}(\mathcal{N})$. Observe that for any $E \in \mathcal{A}(\mathcal{N})$, we have the positive head reduction

$$\begin{aligned}&\langle \tilde{\mu}[\mathsf{Fall}^{j<N}[\alpha](x).c] \| E \rangle\\&\mapsto_+ \langle \tilde{\mu}[\mathsf{Fall}^{i<N}[\alpha].c\{i/j, \tilde{\mu}[\mathsf{Fall}^{j<i}[\alpha](x).c]/x\}] \| E \rangle\end{aligned}$$

And since $Head(\mathcal{A}(\mathcal{N})) \sqsubseteq \mathcal{A}(\mathcal{N}) = \mathcal{A}(\mathcal{N})^{\perp}$, it suffices to show that

$$\tilde{\mu}[\mathsf{Fall}^{i<N}[\alpha].c\{i/j, \tilde{\mu}[\mathsf{Fall}^{j<i}[\alpha](x).c]/x\}] \in \mathcal{A}(\mathcal{N})$$

This follows from our assumption about $\mathcal{A}(\mathcal{N})$ so long as

$$\begin{aligned}&\forall M \llbracket < N \rrbracket_{\mathcal{H}}\ \mathcal{M}, E \in \mathcal{B}(\mathcal{M}).\\&c\{M/j, E/\alpha, \tilde{\mu}[\mathsf{Fall}^{j<M}[\alpha](x).c]/x\} \in \perp\!\!\!\perp\end{aligned}$$

which follows from our assumption about c so long as for any $M \llbracket < N \rrbracket_{\mathcal{H}}\ \mathcal{M}$,

$$\tilde{\mu}[\mathsf{Fall}^{j<M}[\alpha](x).c] \in \mathcal{A}(\mathcal{M})$$

which we show by the inductive hypothesis.

Now, suppose that $O \llbracket < M \rrbracket_{\mathcal{H}}\ \mathcal{O}$, $E \in \mathcal{B}(\mathcal{O})$, and $V \in \mathcal{A}(\mathcal{O})$. By unfolding the definitions of $\llbracket < M \rrbracket_{\mathcal{H}}$ and $\llbracket < N \rrbracket_{\mathcal{H}}$, we have:

- $M \llbracket < N \rrbracket_{\mathcal{H}}\mathcal{M} \equiv \exists M' \in Type.M \twoheadrightarrow_\beta M' \wedge \llbracket M' \rrbracket_{\mathcal{H}} \leq \mathcal{M} < \llbracket N \rrbracket_{\mathcal{H}}$,
- $O \llbracket < M \rrbracket_{\mathcal{H}}\mathcal{O} \equiv \exists O' \in Type.O \twoheadrightarrow_\beta O' \wedge \llbracket O' \rrbracket_{\mathcal{H}} \leq \mathcal{O} < \llbracket M \rrbracket_{\mathcal{H}}$,

Note that forward reduction of syntactic types cannot change the ordinal value of their interpretation.

Lemma 33. *If $\llbracket M \rrbracket_{\mathcal{H}} \in \mathbb{O}$ and $M \twoheadrightarrow_\beta M'$ then $\llbracket M \rrbracket_{\mathcal{H}} = \llbracket M' \rrbracket_{\mathcal{H}} \in \mathbb{O}$.*

Proof. By induction on the reductions $M \twoheadrightarrow_\beta M'$ and induction on the source term to find the redex. □

Therefore, $\llbracket O' \rrbracket_{\mathcal{H}} \leq \mathcal{O} < \llbracket M \rrbracket_{\mathcal{H}} \leq \mathcal{M} < \llbracket N \rrbracket_{\mathcal{H}}$, and so $O \llbracket < N \rrbracket_{\mathcal{H}}\mathcal{O}$ as well.

We can now use the assumption on the command c again to obtain $c\{O/j, E/\alpha, V/x\} \in \perp\!\!\!\perp$. Thus it follows that

$$\forall O \llbracket < M \rrbracket_{\mathcal{H}}\mathcal{O}, E \in \mathcal{B}(\mathcal{O}), V \in \mathcal{A}(\mathcal{O}).c\{O/j, E/\alpha, V/x\} \in \perp\!\!\!\perp$$

Applying the inductive hypothesis now grants us that

$$\tilde{\mu}[\mathsf{Fall}^{j<M}[\alpha](x).c] \in \mathcal{A}(\mathcal{M})$$

thus completing the proof. □

As a corollary, note that so long as $\mathcal{H}(\mathsf{Descend}(\mathcal{N},\mathcal{B}))$ is a reducibility candidate for all $\mathcal{N} \in \mathbb{O}$ and $\mathcal{B} : \mathbb{O} \to CR$ where $\langle\!\langle \mathsf{Descend}(\mathcal{N},\mathcal{B}) \rangle\!\rangle_{\mathcal{H}} \sqsubseteq \mathcal{H}(\mathsf{Descend}(\mathcal{N},\mathcal{B}))$, then the function $\lambda\mathcal{N} \in \mathbb{O}.\mathcal{H}(\mathsf{Descend}(\mathcal{N},\mathcal{B}))$ meets the criteria of Lemma 32, so it contains all the matching recursive case abstractions

$$\tilde{\mu}[\mathsf{Fall}^{j<N}(x)[\alpha].c] \in \mathsf{Descend}(\mathcal{N},\mathcal{B})$$

meeting the specified criteria for every $N \llbracket \mathsf{Ord} \rrbracket_{\mathcal{H}}\ \mathcal{N}$.

The co-data type constructor Ascend follows analogously by duality.

F.5 Deflate and Inflate

Similarly, we can compute the definition of Deflate.

$$Cons_{\mathcal{H}}(\mathsf{Deflate}(\mathcal{A})) \triangleq \{\mathsf{Down}^M(V) | M \llbracket \mathsf{Ix} \rrbracket_{\mathcal{H}}\ \mathcal{M}, V \in \mathcal{A}(\mathcal{M})\}$$

All the usual properties for a data type like Deflate hold, and in addition we have the recursive form of case abstraction.

Lemma 34 (Looping)**.** *Suppose $\mathcal{H}$ is plausible and $\mathcal{A} : \pi_1(\llbracket \mathsf{Ix} \rrbracket_{\mathcal{H}}) \to CR$. Given c_0, c_1 such that*

$$\begin{aligned}&\forall V \in \mathcal{A}(\mathcal{H}(0)).c_0\{V/x\} \in \perp\!\!\!\perp\\&\forall N \llbracket \mathsf{Ix} \rrbracket_{\mathcal{H}}\ \mathcal{N}, V \in \mathcal{A}(\mathcal{H}(+1(\mathcal{N}))), E \in \mathcal{A}(\mathcal{N}).\\&\quad c_1\{N/j, V/x, E/\alpha\} \in \perp\!\!\!\perp\end{aligned}$$

then $E = \tilde{\mu}[\mathsf{Down}^{0:\mathsf{Ix}}(x).c | \mathsf{Down}^{j+1:\mathsf{Ix}}(x)[\alpha].c_1] \in \langle\!\langle \mathsf{Deflate}(\mathcal{A}) \rangle\!\rangle_{\mathcal{H}}$.

Proof. Note that $\langle\!\langle \mathsf{Deflate}(\mathcal{A}) \rangle\!\rangle_{\mathcal{H}} = \langle\!\langle \mathsf{Deflate}(\mathcal{A}) \rangle\!\rangle_{\mathcal{H}}{}^{\perp s}$. Since E is simple, it remains to show that $E \in \mathcal{W}$ and for all $V \in Cons_{\mathcal{H}}(\mathsf{Deflate}(\mathcal{A}))$, $\langle V \| E \rangle \in \perp\!\!\!\perp$.

- $E \in \mathcal{W}$: Note that since $\mathcal{H}$ is plausible, $\mathcal{H}(0), \mathcal{H}(+1(\mathcal{H}(0))) \in \mathbb{O}$. Additionally, $0 \sim_{\mathcal{H}} \mathcal{H}(0)$ and $0+1 \sim_{\mathcal{H}} \mathcal{H}(+1(\mathcal{H}(0)))$ by definition. Therefore, $\mathcal{A}(\mathcal{H}(0)), \mathcal{A}(\mathcal{H}(+1(\mathcal{H}(0)))) \in CR$, and so both contain the (co-)variables by Lemma 2. Therefore, $c_0 \in \perp\!\!\!\perp$ and $c_1\{0/j\} \in \perp\!\!\!\perp$, and so $c_1 \in \perp\!\!\!\perp$ by Lemma 20. Thus, E is strongly normalizing so $E \in \mathcal{W}$.
- For all $V \in Cons_{\mathcal{H}}(\mathsf{Deflate}(\mathcal{A}))$, $\langle V \| E \rangle \in \perp\!\!\!\perp$: Note that $V = \mathsf{Down}^N(V')$ for some $N \sim_{\mathcal{H}} \mathcal{N}$ and $V' \in \mathcal{A}(\mathcal{N})$. We proceed by induction on $N \sim_{\mathcal{H}} \mathcal{N}$.

- $0 \sim_{\mathcal{H}} \mathcal{H}(0)$: Observe that
$$\langle V \| E\rangle \mapsto_0 c_0\{V'/x\} \in \perp\!\!\!\perp$$
so $\langle V \| E\rangle \in \perp\!\!\!\perp$ by Lemma 14.
- $M+1 \sim_{\mathcal{H}} \mathcal{H}(+1(\mathcal{M}))$ for some $M \sim_{\mathcal{H}} \mathcal{M}$: Observe that
$$\langle V \| E\rangle \mapsto_0 \langle \mu\alpha.c_1\{M/j, V/x\} \| \tilde{\mu}y.\langle \mathsf{Down}^M(y) \| E\rangle\rangle$$
By the inductive hypothesis, $\langle \mathsf{Down}^M(V') \| E\rangle \in \perp\!\!\!\perp$ for all $V' \in \mathcal{A}(\mathcal{N})$, so $\tilde{\mu}y.\langle \mathsf{Down}^M(y) \| E\rangle \in \mathcal{A}(\mathcal{N})$ by strong activation (Lemma 11) because $\mathcal{A}(\mathcal{N})$ is a reducibility candidate. Furthermore, for all $E' \in \mathcal{A}(\mathcal{N})$, $c_1\{M/j, V/x, E'/\alpha\} \in \perp\!\!\!\perp$ by assumption on c_1, so we have $\mu\alpha.c_1\{M/j, V/x\}$ for the same reason. Therefore,
$$\langle \mu\alpha.c_1\{M/j, V/x\} \| \tilde{\mu}y.\langle \mathsf{Down}^M(y) \| E\rangle\rangle \in \perp\!\!\!\perp$$
and $\langle V \| E\rangle \in \perp\!\!\!\perp$ as well by Lemma 14. □

The co-data type constructor $\mathsf{Inflate}$ follows analogously by duality.

G. Soundness

We give the following meaning to the sequents

$$\begin{aligned}
[\![\Theta \vdash A : k]\!]_{\mathcal{H}} &\iff (\forall \theta [\![\Theta]\!]_{\mathcal{H}} \mathcal{I}. A\{\theta\}\ [\![k]\!]_{\mathcal{I}}\ [\![A]\!]_{\mathcal{I}}) \\
&\quad \wedge \forall \mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}, [\![A]\!]_{\mathcal{I}} \in [\![k]\!]_{\mathcal{I}} \\
[\![\Theta \vdash k : s]\!]_{\mathcal{H}} &\iff \forall \mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}, [\![k]\!]_{\mathcal{I}} \in [\![s]\!] \\
[\![(\vdash .)]\!]_{\mathcal{H}} &\iff true \\
[\![(\vdash_{\Theta, a:k})]\!]_{\mathcal{H}} &\iff [\![(\vdash_{\Theta})]\!]_{\mathcal{H}} \wedge [\![\Theta \vdash k : \blacksquare]\!]_{\mathcal{H}} \\
[\![(\Gamma \vdash_{\Theta} \Delta)\,\mathbf{seq}\,]\!]_{\mathcal{H}} &\iff [\![\vdash_{\Theta}]\!]_{\mathcal{H}} \\
&\quad \wedge (\forall x : A \in \Gamma. [\![\Theta \vdash A : \star]\!]_{\mathcal{H}}) \\
&\quad \wedge (\forall \alpha : A \in \Delta. [\![\Theta \vdash A : \star]\!]_{\mathcal{H}}) \\
[\![\Theta \vdash A = B : k]\!]_{\mathcal{H}} &\iff \forall \mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}. [\![A]\!]_{\mathcal{I}} = [\![B]\!]_{\mathcal{I}} \in [\![k]\!]_{\mathcal{I}} \\
[\![c : (\Gamma \vdash_{\Theta} \Delta)]\!]_{\mathcal{H}} &\iff \forall \theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}. \forall \gamma \in [\![\Gamma]\!]_{\mathcal{I}}, \delta \in [\![\Delta]\!]_{\mathcal{I}}. \\
&\quad c\{\gamma, \delta, \theta\} \in \perp\!\!\!\perp \\
[\![\Gamma \vdash_{\Theta} v : T \mid \Delta]\!]_{\mathcal{H}} &\iff \forall \theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}. \forall \gamma \in [\![\Gamma]\!]_{\mathcal{I}}, \delta \in [\![\Delta]\!]_{\mathcal{I}}. \\
&\quad v\{\gamma, \delta, \theta\} \in [\![T]\!]_{\mathcal{I}} \\
[\![\Gamma \mid e : T \vdash_{\Theta} \Delta]\!]_{\mathcal{H}} &\iff \forall \theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}. \forall \gamma \in [\![\Gamma]\!]_{\mathcal{I}}, \delta \in [\![\Delta]\!]_{\mathcal{I}}. \\
&\quad e\{\gamma, \delta, \theta\} \in [\![T]\!]_{\mathcal{I}}
\end{aligned}$$

Lemma 35. *Given any $\mathcal{H} \in [\![\mathcal{F}]\!]$:*

1. *if $\Theta \vdash k : s$ and $(\vdash_{\Theta})\,\mathbf{seq}$ are derivable in $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$ then $[\![\Theta \vdash k : s]\!]_{\mathcal{H}}$;*
2. *if $\Theta \vdash A : k$ and $(\vdash_{\Theta, a:k})\,\mathbf{seq}$ are derivable in $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$ then $[\![\Theta \vdash A : k]\!]_{\mathcal{H}}$;*
3. *if $\Theta \vdash A = B : k$ and $(\vdash_{\Theta, a:k})\,\mathbf{seq}$ are derivable in $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$ then $[\![\Theta \vdash A = B : k]\!]_{\mathcal{H}}$;*
4. *if $(\Gamma \vdash_{\Theta} \Delta)\,\mathbf{seq}$ is derivable in $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$ then $[\![(\Gamma \vdash_{\Theta} \Delta)\,\mathbf{seq}\,]\!]_{\mathcal{H}}$.*

Proof. By mutual induction on the typing derivation. In the case where we have two assumption derivations, it is the first one which we take to be decreasing.

- $$\frac{}{\Theta \vdash 0 : \mathsf{Ix}}$$
If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ then $\mathcal{I} \in [\![\mathcal{F}]\!]$ by Lemma 8, so $[\![0]\!]_{\mathcal{I}} \in \mathbb{O}$. Further, $0\{\theta\} = 0 \sim_{\mathcal{I}} \mathcal{I}(0)$.
- $$\frac{\Theta \vdash M : \mathsf{Ix}}{\Theta \vdash M+1 : \mathsf{Ix}}$$
If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ then $\mathcal{I} \in [\![\mathcal{F}]\!]$ by Lemma 8 and since by the inductive hypothesis $[\![M]\!]_{\mathcal{I}} \in [\![\mathsf{Ix}]\!]_{\mathcal{I}}$, we know that $[\![M+1]\!]_{\mathcal{I}} = \mathcal{I}(+1([\![M]\!]_{\mathcal{I}})) \in [\![\mathsf{Ix}]\!]_{\mathcal{I}}$. Further, if $\theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}$ then $M\{\theta\}\ [\![\mathsf{Ix}]\!]_{\mathcal{I}}\ [\![M]\!]_{\mathcal{I}}$ so $M\{\theta\} \sim_{\mathcal{I}} [\![M]\!]_{\mathcal{I}}$ and $(M+1)\{\theta\} \sim_{\mathcal{I}} \mathcal{I}(+1([\![M]\!]_{\mathcal{I}}))$.
- $$\frac{}{\Theta \vdash 0 < \infty}$$
If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ then $\mathcal{I} \in [\![\mathcal{F}]\!]$ by Lemma 8 so this holds by the requirement that $\mathcal{I}(0) < \mathcal{I}(\infty)$ from Definition 6.
- $$\frac{\Theta \vdash M < \infty}{\Theta \vdash M+1 < \infty}$$
If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ then $\mathcal{I} \in [\![\mathcal{F}]\!]$ by Lemma 8 so this holds by the requirement that $\mathcal{I}(+1(\mathcal{M})) < \mathcal{I}(\infty)$ whenever $\mathcal{M} < \mathcal{I}(\infty)$ from Definition 6.
- $$\frac{\Theta \vdash M < N}{\Theta \vdash M < M+1}$$
If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ then $\mathcal{I} \in [\![\mathcal{F}]\!]$ by Lemma 8 so this holds by the requirement that when ever $\mathcal{M} < \mathcal{N}$ we have $\mathcal{M} < \mathcal{I}(+1(\mathcal{M})) \leq \mathcal{N}$ from Definition 6.
- $$\frac{\Theta \vdash M_1 < M_2 \quad \Theta \vdash M_2 < M_3}{\Theta \vdash M_1 < M_3}$$
This holds by the underlying transitivity of the ordering on $\mathbb{O}$.
- $$\frac{a : k' \notin \Theta'}{\Theta, a : k, \Theta' \vdash a : k}$$
By induction on the length of Θ'. The proof of the base case is immediate. In the inductive case if $[\![\Theta, a : k, \Theta' \vdash a : k]\!]_{\mathcal{H}}$ then for any $\mathcal{I} \in [\![\Theta, a : k, \Theta', b : k']\!]_{\mathcal{H}}$ there must exist some $\mathcal{I}' \in [\![\Theta, a : k, \Theta']\!]_{\mathcal{H}}$ that agree on everything except b. Thus, we know that $[\![a]\!]_{\mathcal{I}} = [\![a]\!]_{\mathcal{I}'} \in [\![k]\!]_{\mathcal{I}'} = [\![k]\!]_{\mathcal{I}}$. Similarly, if $\theta\ [\![\Theta, a : k, \Theta', b : k']\!]_{\mathcal{H}}\ \mathcal{I}$ then there exists $\theta'\ [\![\Theta, a : k, \Theta']\!]_{\mathcal{H}}\ \mathcal{I}'$ such that $a\{\theta\} = a\{\theta'\}\ [\![k]\!]_{\mathcal{I}'}\ [\![a]\!]_{\mathcal{I}'} = [\![a]\!]_{\mathcal{I}}$ and so $a\{\theta\}\ [\![k]\!]_{\mathcal{I}}\ [\![a]\!]_{\mathcal{I}}$.
- $$\frac{\Theta, a : k_1 \vdash B : k_2 \quad \Theta \vdash k_2 : \square}{\Theta \vdash \lambda a : k_1.B : k_1 \to k_2}$$
If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ and $\mathcal{A}$ in $[\![k_1]\!]_{\mathcal{I}}$ then $\mathcal{I}\{\mathcal{A}/a\} \in [\![\Theta, a : k_1]\!]_{\mathcal{H}}$ so, by the inductive hypothesis, $[\![B]\!]_{\mathcal{I}\{\mathcal{A}/a\}} \in [\![k_2]\!]_{\mathcal{I}\{\mathcal{A}/a\}}$ and $[\![k_2]\!]_{\mathcal{I}} \in [\![\square]\!]$. Because $\Theta \vdash k_2 : \square$, $a \notin FV(k_2)$, so by extending $[\![k_2]\!]_{\mathcal{I}} = [\![k_2]\!]_{\mathcal{I}\{\mathcal{A}/a\}}$, we have $[\![B]\!]_{\mathcal{I}\{\mathcal{A}/a\}} \in [\![k_2]\!]_{\mathcal{I}}$ meaning $[\![\lambda a : k_1.B]\!]_{\mathcal{I}} \in [\![k_1 \to k_2]\!]_{\mathcal{I}}$. Now, suppose $\theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}$, then for any $A\{\theta\}\ [\![k_1]\!]_{\mathcal{I}}\ \mathcal{A}$ we have that $\theta\{A\{\theta\}/a\}\ [\![\Theta, a : k_1]\!]_{\mathcal{I}}\ \mathcal{I}\{\mathcal{A}/a\}$ and so it follows that as well $B\{\theta, A\{\theta\}/a\}\ [\![k_2]\!]_{\mathcal{I}\{\mathcal{A}/a\}}\ [\![B]\!]_{\mathcal{I}\{\mathcal{A}/a\}}$. Again, we get $B\{\theta, A\{\theta\}/a\}\ [\![k_2]\!]_{\mathcal{I}}\ [\![B]\!]_{\mathcal{I}\{\mathcal{A}/a\}}$ by extending $[\![k_2]\!]_{\mathcal{I}} \in [\![\square]\!]$. Furthermore, since $[\![k_2]\!]_{\mathcal{I}} \in [\![\square]\!]$, we know that because $(\lambda a : k_1.B)\ A \to_{\beta} B\{A/a\}$, then we also have that $(\lambda a : k_1.B\{\theta\})\ A\{\theta\}\ [\![k_2]\!]_{\mathcal{I}}\ [\![B]\!]_{\mathcal{I}\{\mathcal{A}/a\}}$ Therefore, we have $(\lambda a : k_1.B)\{\theta\}\ [\![k_1 \to k_2]\!]_{\mathcal{I}}\ [\![\lambda a : k.B]\!]_{\mathcal{I}}$.
- $$\frac{\Theta \vdash A : k_1 \to k_2 \quad \Theta \vdash B : k_1}{\Theta \vdash A\ B : k_2}$$
If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ then, by inductive hypothesis $[\![A]\!]_{\mathcal{I}} : [\![k_1]\!]_{\mathcal{I}} \to [\![k_2]\!]_{\mathcal{I}}$ and $[\![B]\!]_{\mathcal{I}} \in [\![k_1]\!]_{\mathcal{I}}$ so $[\![A\ B]\!]_{\mathcal{I}} \in [\![k_2]\!]_{\mathcal{I}}$. By the same token, if $\theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}$ then $B\{\theta\}\ [\![k_1]\!]\ [\![B]\!]_{\mathcal{I}}$ so $(A\ B)\{\theta\}\ [\![k_2]\!]\ [\![A\ B]\!]_{\mathcal{I}}$.
- $$\frac{\Theta \vdash M < N \quad \Theta \vdash N : \mathsf{Ord}}{\Theta \vdash M : \mathsf{Ord}}$$
If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ then by inductive hypothesis $[\![M]\!]_{\mathcal{I}} \in [\![< N]\!]_{\mathcal{I}}$ which means that $[\![M]\!]_{\mathcal{I}} \in \mathbb{O}$ so $[\![M]\!]_{\mathcal{I}} \in [\![\mathsf{Ord}]\!]_{\mathcal{I}}$. If $\theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}$

then by inductive hypothesis, $M\{\theta\}\ [\![< N]\!]_{\mathcal{I}}\ [\![M]\!]_{\mathcal{I}}$ meaning that there is a $M\{\theta\} \twoheadrightarrow_\beta M'$ such that $[\![M']\!]_{\mathcal{I}} \leq [\![M]\!]_{\mathcal{I}} < [\![N]\!]_{\mathcal{I}}$ thus $M\{\theta\}\ [\![\mathsf{Ord}]\!]_{\mathcal{I}}\ [\![M]\!]_{\mathcal{I}}$.
Note that the use of the inductive hypothesis here is slightly non trivial and so requires explanation. We know that $(\vdash_{\Theta,a:\mathsf{Ord}})\,\mathbf{seq}$ is derivable and thus $(\vdash_\Theta)\,\mathbf{seq}$ is derivable. Further, since $\Theta \vdash N : \mathsf{Ord}$ is derivable so is $\Theta \vdash< N : \square$ and so $(\vdash_{\Theta,a<N})\,\mathbf{seq}$.

- $$\frac{}{\Theta \vdash \infty : \mathsf{Ord}}$$
  By the requirement on $\mathcal{I} \in [\![\mathcal{F}]\!]$ that $\mathcal{I}(\infty) \in \mathbb{O}$.
- $$\frac{\overrightarrow{\Theta \vdash A : k}}{\Theta \vdash \mathsf{F}(\overrightarrow{A}) : \star}$$
  For any $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ we know that $\mathcal{I} \in [\![\mathcal{F}]\!]$ and since, by inductive hypothesis, $\overrightarrow{[\![A]\!]_{\mathcal{I}} \in [\![k]\!]_{\mathcal{I}}}$ that means $[\![\mathsf{F}(\overrightarrow{A})]\!]_{\mathcal{I}} = \mathcal{I}(\mathsf{F}(\overrightarrow{[\![A]\!]_{\mathcal{I}}})) \in CR$ so $[\![\mathsf{F}(\overrightarrow{A})]\!]_{\mathcal{I}} \in [\![\star]\!]_{\mathcal{I}}$. Now, if $\theta\ [\![\Theta]\!]\ \mathcal{I}$ then $\mathsf{F}(\overrightarrow{A})\{\theta\}\ [\![\star]\!]_{\mathcal{I}}\ [\![\mathsf{F}(\overrightarrow{A})]\!]_{\mathcal{I}}$ since $[\![\star]\!]$ is the total relation.
- $$\frac{\Theta \vdash k : \square}{\Theta \vdash k : \blacksquare}$$
  For any $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$, Observe that $[\![\blacksquare]\!]$ contains everything in $[\![\square]\!]$, so the inductive hypothesis $[\![\Theta \vdash k : \square]\!]_{\mathcal{I}}$ implies $[\![\Theta \vdash k : \blacksquare]\!]_{\mathcal{I}}$.
- $$\frac{\Theta \vdash k_1 : \blacksquare \quad \Theta \vdash k_2 : \square}{\Theta \vdash k_1 \to k_2 : \square}$$
  If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ then, by the inductive hypothesis, $[\![k_1]\!]_{\mathcal{I}} \in [\![\blacksquare]\!]$ and $[\![k_2]\!]_{\mathcal{I}} \in [\![\square]\!]$, so for any $A'\ B\ [\![k_2]\!]_{\mathcal{I}}\ \mathcal{A}(\mathcal{B})$, if $A \twoheadrightarrow_\beta A'$ then $A\ B\ [\![k_2]\!]_{\mathcal{I}}\ \mathcal{S}(\mathcal{B})$ as well. Therefore $[\![k_1 \to k_2]\!]_{\mathcal{I}} \in [\![\square]\!]$ since $A'[\![k_1 \to k_2]\!]_{\mathcal{I}}\mathcal{A}$ and $A \twoheadrightarrow_\beta A'$ implies that $A[\![k_1 \to k_2]\!]_{\mathcal{I}}\mathcal{A}$.
- $$\frac{}{\Theta \vdash \star : \square}$$
  Note that for any $\mathcal{I}$, syntactic type A, and $CR\ \mathcal{B}$, $A\ [\![\star]\!]_{\mathcal{I}}\ \mathcal{B}$, so $[\![\star]\!]_{\mathcal{I}} \in [\![\square]\!]$.
- $$\frac{}{\Theta \vdash \mathsf{Ix} : \blacksquare}$$
  Immediate.
- $$\frac{}{\Theta \vdash \mathsf{Ord} : \square}$$
  If $\mathcal{I} \in [\![\Theta]\!]$, then note that by definition of $[\![\mathsf{Ord}]\!]_{\mathcal{I}}$, $M'\ [\![\mathsf{Ord}]\!]_{\mathcal{I}}\ \mathcal{M}$ and $M \twoheadrightarrow_\beta M'$ implies that $M\ [\![\mathsf{Ord}]\!]_{\mathcal{I}}\ \mathcal{M}$ as well. Therefore, $[\![\mathsf{Ord}]\!]_{\mathcal{I}} \in [\![\square]\!]$.
- $$\frac{\Theta \vdash N : \mathsf{Ord}}{\Theta \vdash (< N) : \square}$$
  We have $\Theta \vdash \mathsf{Ord} : \blacksquare$ so $(\vdash_\Theta)\,\mathbf{seq}$ allows us to prove $(\vdash_{\Theta,a:\mathsf{Ord}})\,\mathbf{seq}$. Thus, by induction we know that $[\![\Theta \vdash N : \mathsf{Ord}]\!]_{\mathcal{H}}$. So, for any $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$ we have $[\![N]\!]_{\mathcal{I}} \in \mathbb{O}$. Furthermore, for any $M'\ [\![< N]\!]_{\mathcal{I}}\ \mathcal{M}$ where $M \twoheadrightarrow_\beta M'$, then $M\ [\![< N]\!]_{\mathcal{I}}\ \mathcal{M}$ as well, so $[\![< N]\!]_{\mathcal{I}} \in [\![\square]\!]$.
- $$\frac{\Theta, a : k_1 \vdash B : k_2 \quad \Theta \vdash A : k_1 \quad \Theta \vdash k_1 : \blacksquare}{\Theta \vdash B\{A/a\} = (\lambda a : k_1.B)\ A : k_2}$$
  If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$, by the inductive hypothesis, we know that $[\![A]\!]_{\mathcal{I}} \in [\![k_1]\!]_{\mathcal{I}}$ and $\mathcal{I}\{[\![A]\!]_{\mathcal{I}}/a\} \in [\![\Theta, a : k_1]\!]$ so $[\![B]\!]_{\mathcal{I}\{[\![A]\!]_{\mathcal{I}}/a\}} = [\![B\{A/a\}]\!]_{\mathcal{I}} \in [\![k_2]\!]\mathcal{I}$ by Lemma 22 as well. Furthermore, $[\![(\lambda a : k_1.B)\ A]\!]_{\mathcal{I}} = [\![B]\!]_{\mathcal{I}\{[\![A]\!]_{\mathcal{I}}/a\}} = [\![B\{A/a\}]\!]_{\mathcal{I}}$.
  Note that to invoke the inductive hypothesis on $\Theta, a : k_1 \vdash B : k_2$, we need $(\vdash_{\Theta,a:k_1,b:k_2})\,\mathbf{seq}$, which is derivable from the given $(\vdash_{\Theta,b:k_2})\,\mathbf{seq}$ and the premise $\Theta \vdash k_1 : \blacksquare$.
- $$\frac{\Theta \vdash B : k_1 \to k_2}{\Theta \vdash (\lambda a : k_1.B\ a) = B : k_1 \to k_2}$$
  If $\mathcal{I} \in [\![\Theta]\!]_{\mathcal{H}}$, then by the inductive hypothesis we have that $[\![B]\!]_{\mathcal{I}} \in [\![k_1 \to k_2]\!]_{\mathcal{I}}$. Furthermore, observe that we have $\lambda\mathcal{A} \in [\![k_1]\!]_{\mathcal{I}}.[\![B]\!]_{\mathcal{I}\{\mathcal{A}/a\}}(\mathcal{A}) = \lambda\mathcal{A} \in [\![k_1]\!]_{\mathcal{I}}.[\![B]\!]_{\mathcal{I}}(\mathcal{A})$ by Lemma 22 because $a \notin FV(B)$, so it denotes the same function as $[\![B]\!]_{\mathcal{I}}$.
- $$\frac{\Theta \vdash A : k}{\Theta \vdash A = A : k} \qquad \frac{\Theta \vdash B = A : k}{\Theta \vdash A = B : k}$$
  $$\frac{\Theta \vdash A = B : k \quad \Theta \vdash B = C : k}{\Theta \vdash A = C : k}$$
  $$\frac{\Theta \vdash A = A' : k_1 \to k_2 \quad \Theta \vdash B = B' : k_1 \quad \Theta \vdash k_1 : \blacksquare}{\Theta \vdash A\ B = A'\ B' : k_2}$$
  $$\frac{\Theta, a : k_1 \vdash B = B' : k_2}{\Theta \vdash (\lambda a : k_1.B) = (\lambda a : k_1.B') : k_1 \to k_2}$$
  Immediate by the inductive hypothesis. Note that to apply the inductive hypothesis for the equivalence $\Theta \vdash A\ B = A'\ B' : k_2$, we need the premise $\Theta \vdash k_1 : \blacksquare$ to ensure that $(\vdash_{\Theta,b:k_1})\,\mathbf{seq}$ is derivable.
  Also note that α-equivalence for type-level lambdas is derivable from the $\beta\eta$ equalities along with these rules.
- The rules for $\mathbf{seq}$ are immediate. $\square$

Lemma 36. *If all (co-)data declarations in $\mathcal{F}$ are well-formed, then $\mathcal{F}$ is well-founded.*

Proof. By induction on the order of declarations in $\mathcal{F}$. For each declaration, we extend the order accordingly and use either Lemma 23, Lemma 26, or Lemma 23, depending on which kind of declaration it is, to ensure that the extended order satisfies the conditions in Definition 7. $\square$

Theorem 2. *For any $\mathcal{H} \in [\![\mathcal{F}]\!]$,*

- *If $c : \Gamma \vdash_\Theta \Delta$ and $(\Gamma \vdash_\Theta \Delta)\,\mathbf{seq}$ are derivable in $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$ then $[\![c : \Gamma \vdash_\Theta \Delta]\!]_{\mathcal{H}}$.*
- *If $\Gamma \vdash_\Theta v : A|\Delta$ and $(\Gamma \vdash_\Theta \alpha : A, \Delta)\,\mathbf{seq}$ are derivable in $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$ then $[\![\Gamma \vdash_\Theta v : A|\Delta]\!]_{\mathcal{H}}$.*
- *If $\Gamma|e : A \vdash_\Theta \Delta$ and $(\Gamma, x : A \vdash_\Theta \Delta)\,\mathbf{seq}$ are derivable in $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$ then $[\![\Gamma|e : A \vdash_\Theta \Delta]\!]_{\mathcal{H}}$.*

Proof. By induction on the first argument. In each case, suppose that, $\theta\ [\![\Theta]\!]_{\mathcal{H}}\ \mathcal{I}$ and $\gamma \in [\![\Gamma]\!]_{\mathcal{I}}$ and $\delta \in [\![\Delta]\!]_{\mathcal{I}}$, where Γ and Δ are the typing environments of the concluding sequent. Also, note that for each case, because $\mathcal{H} \in [\![\mathcal{F}]\!]$, we know that $[\![A]\!]_{\mathcal{I}}$ is a reducibility candidate whenever $\Theta \vdash A : \star$ and therefore whenever $x : A \in \Gamma$ or $\alpha : A \in \Delta$ due to the assumption that $(\Gamma \vdash_\Theta \Delta)\,\mathbf{seq}$. First, we have the structural rules that apply for any type.

- Var for type A: Note that $x : A \in \Gamma$ so by definition, $x\{\theta, \gamma, \delta\} \in Val([\![A]\!]_{\mathcal{I}}) \sqsubseteq [\![A]\!]_{\mathcal{I}}$.
- $CoVar$ for type A: Analogous to the previous case by duality.
- Cut for type A: By the inductive hypothesis, $v\{\theta, \gamma, \delta\} \in [\![A]\!]_{\mathcal{I}}$ and $e\{\theta, \gamma, \delta\} \in [\![A]\!]_{\mathcal{I}}$. Furthermore, because $[\![A]\!]_{\mathcal{I}}$ is a reducibility candidate we know $[\![A]\!]_{\mathcal{I}} = [\![A]\!]_{\mathcal{I}}{}^{\perp}$, so $\langle v\{\theta, \gamma, \delta\} \| e\{\theta, \gamma, \delta\}\rangle = \langle v \| e\rangle\{\theta, \gamma, \delta\} \in \perp\!\!\!\perp$.
- Act for type A: By the inductive hypothesis, we know that for all $E \in [\![A]\!]_{\mathcal{I}}$, $c\{\theta, \gamma, E/\alpha/\delta\} \in \perp\!\!\!\perp$. Therefore, $\mu\alpha.c\{\theta, \gamma, \delta\} = (\mu\alpha.c)\{\theta, \gamma, \delta\} \in [\![A]\!]_{\mathcal{I}}$ by strong activation (Lemma 11).

- $CoAct$ for type A: Analogous to the previous case by duality.
- Eq for types $A = B$: hold the soundness of type-level equality from Lemma 35, so that $[\![A]\!]_{\mathcal{I}} = [\![B]\!]_{\mathcal{I}}$.
- $CoEq$ for types $A = B$: Analogous to the previous case by duality.

Next, we have the type-specific left and right introduction rules. Remember that for the active types A in each case, because $\mathcal{H} \in [\![\mathcal{F}]\!]$ we know that $\langle\!\langle A \rangle\!\rangle_{\mathcal{I}} \sqsubseteq [\![A]\!]_{\mathcal{I}}$.

Suppose we have a data type

$$\mathbf{data}\, \mathsf{F}(\vec{a} : k)\, \mathbf{where}$$
$$\overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{X}) | \vec{C}}$$

Then if $\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{a}) | \vec{C}$ is in $\overrightarrow{\mathsf{K} : \vec{B} \vdash_{\overrightarrow{d:k'}} \mathsf{F}(\vec{a}) | \vec{C}}$ We have the rule

$$\frac{\overrightarrow{\Theta \vdash D : k'_i\{\overrightarrow{A/a}\}} \quad \overrightarrow{\Gamma \vdash_\Theta v : B_i\{\overrightarrow{A/a}, \overrightarrow{D/d_i}\} | \Delta} \quad \overrightarrow{\Gamma | e : C_i\{\overrightarrow{A/a}, \overrightarrow{D/d_i}\} \vdash_\Theta \Delta}}{\Gamma \vdash_\Theta \mathsf{K}_i^{\vec{\mathcal{D}}}(\vec{e}, \vec{v}) : \mathsf{F}\ \vec{A} | \Delta}$$

Now supposing that each of $\vec{e}$ and $\vec{v}$ are all (co-)values $\vec{E}$ and $\vec{V}$, by the inductive hypothesis, we know that

$$K_i^{\vec{\mathcal{D}}}(\vec{E}, \vec{V})\{\theta, \gamma, \delta\} \in \langle\!\langle \mathsf{F}(\overrightarrow{[\![A]\!]_{\mathcal{I}}}) \rangle\!\rangle_{\mathcal{I}} \sqsubseteq [\![\mathsf{F}(\overrightarrow{[\![A]\!]_{\mathcal{I}}})]\!]_{\mathcal{I}}$$

Furthermore, since

$$\overrightarrow{[\![B_i]\!]_{\mathcal{I}\{\overrightarrow{[\![A]\!]_{\mathcal{I}}/a}, \overrightarrow{[\![D]\!]_{\mathcal{I}}/d_i}\}}} = \overrightarrow{[\![B_i\{\overrightarrow{A/a}, \overrightarrow{D/d_i}\}]\!]_{\mathcal{I}}}$$
$$\overrightarrow{[\![C_i]\!]_{\mathcal{I}\{\overrightarrow{[\![A]\!]_{\mathcal{I}}/a}, \overrightarrow{[\![D]\!]_{\mathcal{I}}/d_i}\}}} = \overrightarrow{[\![C_i\{\overrightarrow{A/a}, \overrightarrow{D/d_i}\}]\!]_{\mathcal{I}}}$$

are all reducibility candidates, we must have that

$$K_i^{\vec{\mathcal{D}}}(\vec{e}, \vec{v})\{\theta, \gamma, \delta\} \in [\![\mathsf{F}(\overrightarrow{[\![A]\!]_{\mathcal{I}}})]\!]_{\mathcal{I}}$$

in general by unfocalization (Lemma 12).

For the left rule

$$\frac{c_1 : (\Gamma, \overrightarrow{x : B_1} \vdash_{\Theta, \overrightarrow{d_1 : l_1}} \Delta, \overrightarrow{\alpha : C_1})\{\overrightarrow{A/a}\} \quad \ldots}{\Gamma | \tilde{\mu}[\mathsf{K}_1^{\overrightarrow{d_1 : l_1}}(\vec{\alpha}, \vec{x}).c_1 | \ldots]\{\overrightarrow{A/a}\} : \mathsf{F}\ \vec{A} \vdash_\Theta \Delta}$$

For each c_i, we have by inductive hypothesis that for any of θ' $[\![\Theta, \overrightarrow{d_i : l_i}]\!]_{\mathcal{H}}$ $\mathcal{I}'$, $\gamma' \in [\![\Gamma, \overrightarrow{x : B_i}]\!]_{\mathcal{I}'}$ and $\delta' \in [\![\Delta, \overrightarrow{\alpha : C_i}]\!]_{\mathcal{I}'}$, $c_i\{\theta', \gamma', \delta'\} \in \perp\!\!\!\perp$. Observe that given any choice of types $\overrightarrow{D\ [\![k'_i]\!]_{\mathcal{I}}\ \mathcal{D}}$, values $\overrightarrow{V \in [\![B_i]\!]_{\mathcal{I}\{\overrightarrow{\mathcal{D}/d_i}\}}}$ and co-values $\overrightarrow{E \in [\![C_i]\!]_{\mathcal{I}\{\overrightarrow{\mathcal{D}/d_i}\}}}$ then

$$\theta\{\overrightarrow{D/d_i}\}\ [\![\Theta, \overrightarrow{d_i : k'_i}]\!]_{\mathcal{H}}\ \mathcal{I}\{\overrightarrow{\mathcal{D}/d_i}\}$$
$$\gamma\{\overrightarrow{V/x}\} \in [\![\Gamma, \overrightarrow{x : B_i}]\!]_{\mathcal{I}\{\overrightarrow{\mathcal{D}/d_i}\}}$$
$$\delta\{\overrightarrow{E/\alpha}\} \in [\![\Delta, \overrightarrow{\alpha : C_i}]\!]_{\mathcal{I}\{\overrightarrow{\mathcal{D}/d_i}\}}$$

Thus,

$$\begin{aligned} &c_i\{\theta\{\overrightarrow{D/d_i}\}, \gamma\{\overrightarrow{V/x}\}, \delta\{\overrightarrow{E/\alpha}\}\} \\ &= c_i\{\theta, \gamma, \delta\}\{\overrightarrow{V/x}, \overrightarrow{E/\alpha}, \overrightarrow{D/d_i}\} \in \perp\!\!\!\perp \end{aligned}$$

From this we can see that

$$\tilde{\mu}[\mathsf{K}_1^{\overrightarrow{d_1 : k'_1}}(\vec{\alpha}, \vec{x}).c_1\{\theta, \gamma, \delta\} | \ldots] \in \langle\!\langle \mathsf{F}(\vec{A}) \rangle\!\rangle_{\mathcal{I}} \sqsubseteq [\![\mathsf{F}(\vec{A})]\!]_{\mathcal{I}}$$

by Lemma 25.

The rules for non-recursive co-data declarations are analogous by duality. Additionally, (co-)data type declarations defined by noetherian and primitive recursion also follow analogously.

The ordinary rules for Ascend and $\mathsf{Descend}$ are handled by viewing them as user defined co-data types. The remaining rule we need to look at is the recursive case abstraction.

$$\frac{c : \Gamma, x : A\ j \vdash_{\Theta, j<N} \alpha : \mathsf{Descend}(j, A), \Delta}{\Gamma | \tilde{\mu}[\mathsf{Fall}^{j<N}(x)[\alpha].c] : \mathsf{Descend}(N, A) \vdash_\Theta \Delta}$$

So that the inductive hypothesis fits the requirements of Lemma 32. Note that the recursive case abstraction for Ascend follows dually.

The recursive case abstraction rule for $\mathsf{Deflate}$ is

$$\frac{c_1 : \Gamma, x : A\ (j+1) \vdash_{\Theta, j:\mathsf{Ix}} \alpha : A\ j, \Delta \quad c_0 : \Gamma, x : A\ 0 \vdash_\Theta \Delta}{\Gamma | \tilde{\mu}[\mathsf{Down}^{0:\mathsf{Ix}}(x).c_0 | \mathsf{Down}^{j+1:\mathsf{Ix}}(x)[\alpha].c_1] : \mathsf{Deflate}(A) \vdash_\Theta \Delta}$$

and its soundness follows by Lemma 34 and the inductive hypothesis. Note that the recursive case abstraction for $\mathsf{Inflate}$ follows dually. ☐

Definition 9. *The set $(\!|\Theta|\!)$ is the subset of the maps $HType \rightharpoonup \mathcal{U}$ which are "big enough" with respect to Θ. That is*

$$\begin{aligned} (\!|\cdot|\!) &= \{\mathcal{H} : HType \rightharpoonup \mathcal{U}\} \\ (\!|\Theta, i : k|\!) &= \{\mathcal{H} \in (\!|\Theta|\!) \mid \mathcal{H}(i) < \mathcal{H}(j)\} \quad (k = \overrightarrow{k \to} < j) \\ (\!|\Theta, i : k|\!) &= (\!|\Theta|\!) \quad (\textit{otherwise}) \end{aligned}$$

Lemma 37. *For all Θ, $(\!|\Theta|\!)$ is inhabited.*

Proof. By induction. We can always set $\mathcal{H}(0)$ to be the length of Θ, which covers the worst case where we have $\Theta = i_{n-1} < 0, i_{n-2} < i_{n-1}, \ldots, i_0 < i_1$ by assigning 0 to $i_1, \ldots, n-2$ to i_{n-2}, and $n-1$ to i_{n-1}. More specifically, whenever we see a $i < M$ in Θ, we can assign i a value based on its position in Θ. ☐

Lemma 38. *For every Θ and well-founded $\mathcal{F}$, there exists a $\mathcal{H} \in (\!|\Theta|\!)$ such that $\mathcal{H} \in [\![\mathcal{F}]\!]$*

Proof. By composition of the previous lemma with Lemma 17. ☐

Lemma 39. *If $\mathcal{H} \in (\!|\Theta|\!)$ and $\mathcal{H} \in [\![\mathcal{F}]\!]$ then there exists θ $[\![\Theta]\!]_{\mathcal{H}}$ $\mathcal{I}$*

Proof. By induction on Θ. ☐

Corollary 9. *If $\mathcal{H} \in (\!|\Theta|\!)$ and $\mathcal{H} \in [\![\mathcal{F}]\!]$ then $[\![c : (\Gamma \vdash_\Theta \Delta)]\!]_{\mathcal{H}}$ implies $c \in \perp\!\!\!\perp$. Similarly, $[\![\Gamma \vdash_\Theta v : A | \Delta]\!]_{\mathcal{H}}$ implies $v \in \mathcal{W}$ and $[\![\Gamma | e : A \vdash_\Theta \Delta]\!]_{\mathcal{H}}$ implies $e \in \mathcal{W}$.*

Theorem 3. *If $\mathcal{F}$ is well-founded, then*

1. *If $c : \Gamma \vdash_\Theta \Delta$ and $(\Gamma \vdash_\Theta \Delta)$ **seq** are derivable in $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$, then c is strongly normalizing in the $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$-calculus.*
2. *If $\Gamma \vdash_\Theta v : A | \Delta$ and $(\Gamma \vdash_\Theta \alpha : A, \Delta)$ **seq** are derivable in $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$, then v is strongly normalizing in the $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$-calculus.*
3. *If $\Gamma | e : A \vdash_\Theta \Delta$ and $(\Gamma \vdash_\Theta \Delta)$ **seq** are derivable in $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$, then e is strongly normalizing in the $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$-calculus.*

Proof. Since there exists $\mathcal{H} \in (\!|\Theta|\!)$ and $\mathcal{H} \in [\![\mathcal{F}]\!]$ we have $[\![c : (\Gamma \vdash_\Theta \Delta)]\!]_{\mathcal{H}}$ by Theorem 2 and we know that $c : (\Gamma \vdash_\Theta \Delta)$ implies that $c \in \perp\!\!\!\perp$ by the previous corollary. The cases for (co-)terms follows analogously. ☐

Note that type-erasure preserves strong normalization precisely because the type-erased rewriting theory is strictly weaker than rewriting the pre-erased programs.

Lemma 40. *If c is strongly normalizing in the $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$-calculus, then $Erase(c)$ is strongly normalizing in the type-erased $\mu\tilde{\mu}_{\mathcal{S}}^{\mathcal{F}}$-calculus, and similarly for (co-)terms.*

Proof. The $Erase$ operation removes all erasable types, with kinds inhabiting $\Box$, from commands and (co-)terms. In particular, we remove the type-level content of constructors $\mathsf{K}^{\vec{D}}(\vec{e}, \vec{v})$ and patterns $\mathsf{K}^{\overrightarrow{d:k}}(\vec{\alpha}, \vec{x})$, leaving only those types with the non-erasable kind lx. Furthermore, because the caveat for reducing inside the branches of a case abstraction are strictly more limiting in the type-erased $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus, we know that

- If $Erase(c) \to c'$ in the type-erased $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus, there is a c'' such that $Erase(c'') = c'$ and $c \to c''$ in the $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus.
- If $Erase(v) \to v'$ in the type-erased $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus, there is a v'' such that $Erase(v'') = v'$ and $v \to v''$ in the $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus.
- If $Erase(e) \to e'$ in the type-erased $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus, there is a e'' such that $Erase(e'') = e'$ and $e \to e''$ in the $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus.

Therefore, because the chosen c, v, e are strongly normalizing in the $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus, their erasure $Erase(c), Erase(v), Erase(e)$ must also be strongly normalizing in the type-erased $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus. $\Box$

H. Natural Deduction Embedding

To show that the natural deduction calculus for effect-free functional programs is strongly normalizing, we demonstrate that:

1. the translation into the call-by-name $\mu\tilde{\mu}_{\mathcal{N}}$-calculus is type-preserving for well-typed terms,
2. each reduction of a natural deduction term corresponds to at least one reduction in $\mu\tilde{\mu}_{\mathcal{N}}$.

Lemma 41. *If $\Gamma \vdash_\Theta v : A$ is derivable then $\Gamma \vdash_\Theta v^\flat : A|$ is derivable.*

Proof. By induction on the structure of the derivation for $\Gamma \vdash_\Theta v : A$. $\Box$

Note that $\twoheadrightarrow^+$ denotes the transitive, but *not* reflexive, closure of $\to$.

Lemma 42. *If $v \to v'$ in the natural deduction calculus then $v^\flat \twoheadrightarrow^+ v'^\flat$.*

Proof. By cases on the reductions in the natural deduction calculus. Note that $v^\flat\{v'^\flat/x\} = (v\{v'/x\})^\flat$ and $v^\flat\{B/b\} = (v\{B/b\})^\flat$.

- $\{\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}] \Rightarrow v' | \ldots\}.\,\mathsf{H}^{\vec{B}}[\vec{v}] \to v'\{\overrightarrow{B/b}, \overrightarrow{v/x}\}$

$$\begin{aligned}
&\{\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}] \Rightarrow v' | \ldots\}.\,\mathsf{H}^{\vec{B}}[\vec{v}]^\flat\\
&= \mu\alpha.\langle \mu(\mathsf{H}^{\overrightarrow{b:k}}[\vec{x}, \beta].\langle v'^\flat \| \beta\rangle | \ldots) \| \mathsf{H}^{\vec{B}}[\vec{v^\flat}, \alpha]\rangle\\
&\to \mu\alpha.\langle v'^\flat\{\overrightarrow{B/b}, \overrightarrow{v^\flat/x}\} \| \alpha\rangle\\
&\to v'^\flat\{\overrightarrow{B/b}, \overrightarrow{v^\flat/x}\}\\
&= (v'\{\overrightarrow{B/b}, \overrightarrow{v/x}\})^\flat
\end{aligned}$$

- **case** $\mathsf{K}^{\vec{B}}(\vec{v})$ **of** $\mathsf{K}^{\overrightarrow{b:k}}(\vec{x}) \Rightarrow v' | \ldots \to v'\{\overrightarrow{B/b}, \overrightarrow{v/x}\}$

$$\begin{aligned}
&\textbf{case } \mathsf{K}^{\vec{B}}(\vec{v}) \textbf{ of } \mathsf{K}^{\overrightarrow{b:k}}(\vec{x}) \Rightarrow v' | \ldots^\flat\\
&= \mu\alpha.\langle \mathsf{K}^{\vec{B}}(\vec{v^\flat}) \| \tilde{\mu}[\mathsf{K}^{\overrightarrow{b:k}}(\vec{x}).\langle v'^\flat \| \alpha\rangle | \ldots]\rangle\\
&\to \mu\alpha.\langle v'^\flat\{\overrightarrow{B/b}, \overrightarrow{v^\flat/x}\} \| \alpha\rangle\\
&\to v'^\flat\{\overrightarrow{B/b}, \overrightarrow{v^\flat/x}\}\\
&= (v'\{\overrightarrow{B/b}, \overrightarrow{v/x}\})^\flat
\end{aligned}$$

- $\{\mathsf{Rise}^{j<N}(x) \Rightarrow v\}$
$\to \{\mathsf{Rise}^{i<N}(x) \Rightarrow v\{i/j, \{\mathsf{Rise}^{j<i}(x) \Rightarrow v\}/x\}\}$

$$\begin{aligned}
&\{\mathsf{Rise}^{j<N}(x) \Rightarrow v\}^\flat\\
&= \mu(\mathsf{Rise}^{j<N}[\alpha](x).\langle v^\flat \| \alpha\rangle)\\
&\to \mu(\mathsf{Rise}^{i<N}[\alpha].\langle v^\flat\{i/j, \mu(\mathsf{Rise}^{j<i}[\alpha](x).\langle v^\flat \| \alpha\rangle)/x\} \| \alpha\rangle)\\
&= \mu(\mathsf{Rise}^{i<N}[\alpha].\langle v^\flat\{i/j, \{\mathsf{Rise}^{j<i}(x) \Rightarrow v\}^\flat/x\} \| \alpha\rangle)\\
&= \mu(\mathsf{Rise}^{i<N}[\alpha].\langle (v\{i/j, \{\mathsf{Rise}^{j<i}(x) \Rightarrow v\}/x\})^\flat \| \alpha\rangle)\\
&= \{\mathsf{Rise}^{i<N} \Rightarrow v\{i/j, \{\mathsf{Rise}^{j<i}(x) \Rightarrow v\}/x\}\}^\flat
\end{aligned}$$

- $\{\mathsf{Up}^0 \Rightarrow v_0 | \,\mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\,\mathsf{Up}^0 \to v_0$

$$\begin{aligned}
&(\{\mathsf{Up}^0 \Rightarrow v_0 | \,\mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\,\mathsf{Up}^0)^\flat\\
&= \mu\alpha.\langle \mu(\mathsf{Up}^0[\beta].\langle v_0^\flat \| \beta\rangle | \mathsf{Up}^{j+1}[\beta](x).\langle v_1^\flat \| \beta\rangle) \| \mathsf{Up}^0[\alpha]\rangle\\
&\to \mu\alpha.\langle v_0^\flat \| \alpha\rangle\\
&\to v_0^\flat
\end{aligned}$$

- $\{\mathsf{Up}^0 \Rightarrow v_0 | \,\mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\mathsf{Up}^{M+1}$
$\to v_1\{M/j, \{\mathsf{Up}^0 \Rightarrow v_0 | \,\mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\mathsf{Up}^M/x\}$

Let V and $V^\flat$ be

$$\begin{aligned}
V &= \{\mathsf{Up}^0 \Rightarrow v_0 | \,\mathsf{Up}^{j+1}(x) \Rightarrow v_1\}\\
V^\flat &= \mu(\mathsf{Up}^0[\beta].\langle v_0^\flat \| \beta\rangle | \mathsf{Up}^{j+1}[\beta](x).\langle v_1^\flat \| \beta\rangle)
\end{aligned}$$

in the following:

$$\begin{aligned}
&(\{\mathsf{Up}^0 \Rightarrow v_0 | \,\mathsf{Up}^{j+1}(x) \Rightarrow v_1\}.\,\mathsf{Up}^{M+1})^\flat\\
&= \mu\alpha.\langle V^\flat \| \mathsf{Up}^{M+1}[\alpha]\rangle\\
&\to \mu\alpha.\langle \mu\beta.\langle V^\flat \| \mathsf{Up}^M[\beta]\rangle \| \tilde{\mu}x.\langle v_1^\flat\{M/j\} \| \alpha\rangle\rangle\\
&\to \mu\alpha.\langle v_1^\flat\{M/j, \mu\beta.\langle V^\flat \| \mathsf{Up}^M[\beta]\rangle/x\} \| \alpha\rangle\\
&\to v_1^\flat\{M/j, \mu\beta.\langle V^\flat \| \mathsf{Up}^M[\beta]\rangle/x\}\\
&= v_1^\flat\{M/j, (V.\,\mathsf{Up}^M)^\flat/x\}\\
&= (v_1\{M/j, V.\,\mathsf{Up}^M\ /x\})^\flat
\end{aligned}$$

- **loop** $\mathsf{Down}^0(v)$ **of** $\mathsf{Down}^0(x) \Rightarrow v_0 | \,\mathsf{Down}^{j+1}(x) \Rightarrow v_1$
$\to v_0$

$$\begin{aligned}
&(\textbf{loop } \mathsf{Down}^0(v) \textbf{ of } \mathsf{Down}^0(x) \Rightarrow v_0 | \,\mathsf{Down}^{j+1}(x) \Rightarrow v_1)^\flat\\
&= \mu\alpha.\langle \mathsf{Down}^0(v^\flat) \| \tilde{\mu}[\mathsf{Down}^0(x).\langle v_0^\flat \| \alpha\rangle | \mathsf{Down}^{j+1}(x)[\beta].\langle v_1^\flat \| \beta\rangle]\rangle\\
&\to \mu\alpha.\langle v_0^\flat\{v^\flat/x\} \| \alpha\rangle\\
&\to v_0^\flat\{v^\flat/x\}\\
&= (v_0\{v/x\})^\flat
\end{aligned}$$

- **loop** $\mathsf{Down}^{M+1}(v)$ **of** $\mathsf{Down}^0(x) \Rightarrow v_0 | \,\mathsf{Down}^{j+1}(x) \Rightarrow v_1$
$\to$ **loop** $\mathsf{Down}^M(v_1\{M/j, v/x\})$ **of**
$\mathsf{Down}^0(x) \Rightarrow v_0$
$| \,\mathsf{Down}^{j+1}(x) \Rightarrow v_1$

Let E_α be

$$\tilde{\mu}[\mathsf{Down}^0(x).\langle v_0^\flat \| \alpha\rangle | \mathsf{Down}^{j+1}(x)[\beta].\langle v_1^\flat \| \beta\rangle]$$

in the following:

$$
\begin{aligned}
&(\mathbf{loop}\ \mathsf{Down}^{M+1}(v)\ \mathbf{of}\ \mathsf{Down}^{0}(x) \Rightarrow v_0 |\ \mathsf{Down}^{j+1}(x) \Rightarrow v_1)^\flat \\
&= \mu\alpha.\langle \mathsf{Down}^{M+1}(v^\flat) \| E_\alpha \rangle \\
&\to \mu\alpha.\langle \mu\beta.\langle v_1^\flat\{M/j, v^\flat/x\} \| \beta\rangle \| \tilde{\mu}y.\langle \mathsf{Down}^{M}(x) \| E_\alpha \rangle\rangle \\
&\to \mu\alpha.\langle \mathsf{Down}^{M}(\mu\beta.\langle v_1^\flat\{M/j, v^\flat/x\} \| \beta\rangle) \| E_\alpha \rangle \\
&\to \mu\alpha.\langle \mathsf{Down}^{M}(v_1^\flat\{M/j, v^\flat/x\}) \| E_\alpha \rangle \\
&= \mu\alpha.\langle \mathsf{Down}^{M}(v_1\{M/j, v/x\})^\flat \| E_\alpha \rangle \\
&= (\mathbf{case}\ \mathsf{Down}^{M}(v_1\{M/j, v/x\})\ \mathbf{of} \\
&\qquad \mathsf{Down}^{0}(x) \Rightarrow v_0 \\
&\quad\ |\ \mathsf{Down}^{j+1}(x) \Rightarrow v_1)^\flat
\end{aligned}
$$

Cases where reduction occurs inside of a larger context follow from compositionality of the translation $(-)^\flat$. □

Theorem 4. *If* $\Gamma \vdash_\Theta v : A$ *and* $(\Gamma \vdash_\Theta \alpha : A)\,\mathbf{seq}$ *are derivable then* v *is strongly normalizing.*

Proof. By Lemma 41 we know that $\Gamma \vdash_\Theta v^\flat : A|$, and so by Theorem 3 $v^\flat$ is a strongly normalizing term of the $\mu\tilde{\mu}_\mathcal{N}$-calculus. Now, suppose that there is an infinite reduction path from v. By applying Lemma 42 over the steps of this infinite reduction path from v and composing them together, we obtain an infinite reduction path from $v^\flat$, which is a contradiction. Therefore, there is no infinite reduction path from v, so v is also strongly normalizing. □

References

[1] F. Barbanera and S. Berardi. A symmetric lambda calculus for "classical" program extraction. In *TACS '94*, pages 495–515, 1994.

[2] P.-L. Curien and G. Munch-Maccagnoni. The duality of computation under focus. *Theoretical Computer Science*, pages 165–181, 2010.

[3] P. Di Gianantonio and M. Miculan. A unifying approach to recursive and co-recursive definitions. In *TYPES*, volume 2646 of *LNCS*. 2003.

[4] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, 1989.

[5] S. Lengrand and A. Miquel. Classical Fω, orthogonality and symmetric candidates. *Annals of Pure and Applied Logic*, 153(1):3–20, 2008.

[6] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87*, 1987.

[7] G. Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, pages 409–423. Springer, 2009.

[8] A. Pitts. A note on logical relations between semantics and syntax. *Logic Journal of IGPL*, 5(4):589–601, 1997.

[9] N. Zeilberger. On the unity of duality. *Annals of Pure Applied Logic*, 153(1-3):66–96, 2008.