The duality of construction

Paul Downen[1] & Zena M. Ariola[1]

University of Oregon, {pdownen,ariola}@cs.uoregon.edu

Abstract. We explore the duality of construction and deconstruction in the presence of different evaluation strategies. We characterize an evaluation strategy by the notion of substitutability, given by defining what is a value and a co-value, and we present an equational theory that takes the strategy as a parameter. The theory may be extended with new logical connectives, in the form of user-defined data and co-data types, which are duals of one another. Finally, we explore a calculus with composite evaluation strategies that allow for more flexibility over evaluation order by mingling multiple primitive strategies within a single program.

1 Introduction

Over two decades ago, Filinski [5] discovered the dual relationship between the call-by-value and call-by-name evaluation strategies by relating programs that produce information with continuations that consume information. Since then, this duality has been studied from the perspective of category theory [5,10] and proof theory [3,11,12]. In particular, the sequent calculus has provided a fruitful foundation for this study, due to the inherent duality in the form of sequent judgments: assumptions act as inputs and conclusions act as outputs. This notion has been formalized [3,11] as foundational calculi which execute at the level of an abstract machine. For example, the inference rule for implication on the left of a sequent is viewed as the typing rule for a call-stack in a Krivine machine.

More recently [13,8,4], polarization in logic has been used as a type-based account of evaluation order, which divides types into two classifications, positive and negative, based on properties of their inference rules. On the one hand, positive types are defined by their rules of introduction, *i.e.,* construction, and are given a call-by-value interpretation. The use of a positively typed value is given by cases over the possible constructions, in the style of data types in functional languages like ML. On the other hand, negative types are defined by their rules of elimination, *i.e.,* observation, and are given a call-by-name interpretation. In order to produce a negatively typed value, we must consider all possible observations, giving us a message-passing programming style. If there is ever an apparent ambiguity on the evaluation order of a program, the type is consulted and the order is determined by considering the type's polarity.

The primary focus on either introduction or elimination divides programs into two parts: concrete programs that are constructed and abstract programs defined by cases. This division describes the behavior of programs as an interaction

between construction and deconstruction, with two dual ways of orienting the roles between a consumer and a producer: data with concrete producers and abstract consumers, and co-data with abstract producers and concrete consumers. In high-level languages, both data and co-data are useful tools for organizing information in programs, and may be interpreted by different evaluation strategies: we may want strictly evaluated terms defined by dynamic dispatch on their observations (as in an object-oriented language), and likewise we may want lazily evaluated terms that are defined by construction (as in Haskell). Polarized logic can account for this behavior by translating a program into one with polarities to provide the desired evaluation order, much like how a continuation-passing style transformation can define the evaluation order for a language.

The goal of this paper is to provide a general account of the data and co-data definitional paradigms along with an equational theory that directly supports evaluation according to different strategies, expressing both the duality between strategies and the two paradigms. A better understanding of the data and (co-)data paradigms may eventually lead to a more suitable foundation for studying the design of languages that contain both functional and object-oriented features. Since the sequent calculus exposes details that appear in abstract machines while still maintaining high-level reasoning principles, it may serve as a bridge between programming languages and their low-level representations, for example as an intermediate language in a compiler.

We develop a sequent calculus that is parameterized by a chosen evaluation strategy, similar to the parametric λ-calculus [9], which guides the notion of substitution in the calculus. The goal of choosing a strategy is to eliminate the fundamental inconsistency of the calculus by eliminating the single point of conflict between producers and consumers. The equational theory is untyped since any conflicts that arise are resolved by the strategy, meaning that we do not need to consult the type of a program during evaluation. We begin by examining the core calculus (Section 3) which expresses the impact of an evaluation strategy on the behavior of a program as a restriction of what may be substituted for a variable. We take notions of call-by-name and call-by-value as our primary examples for characterizing strategies, but also show a characterization of call-by-need and its dual, demonstrating that there are more than two possible strategies.

Atop the core calculus of substitution, we consider functions (Section 4) and describe their behavior in terms of β and η rules. Unlike in the λ-calculus and previous formulations of the sequent calculus [3,11], these same rules apply in every evaluation strategy, and we show that they provide a complete definition of functions. Next, we extend the language with basic data and co-data types (Section 5), illustrating two forms of pairs ($\otimes, \&$) and two forms of sums ($\oplus, \parr$) that correspond to similar concepts in Girard's linear logic [6] and polarized logic [13,4]. As with functions, we give a similar $\beta\eta$ characterization of the basic (co-)data types that does not reference the chosen evaluation strategy, and show that this characterization derives the various "lifting" (ς) rules of Wadler's sequent calculus [11,12]. Finally, we use the common $\beta\eta$ theme in order to present a general notion of user-defined data and co-data types (Section 6) which en-

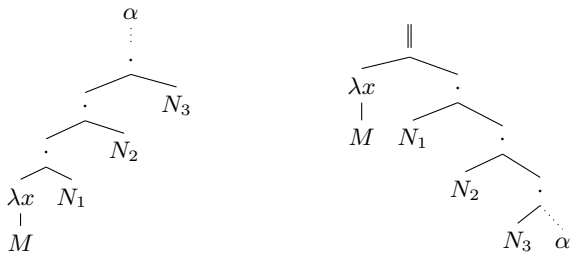
compasses all of the previous types. On the one hand, since we are working in an untyped setting, (co-)data type declarations are used for introducing new ways to form structures and abstractions in a program, independently of a static type system. On the other hand, (co-)data type declarations are inspired by logic and may be seen as describing a static type system for the parametric sequent calculus.

We also consider how to compose several strategies into a single composite strategy (Section 7). This allows a single program to be written with call-by-name and call-by-value parts, or any other combination of two (or more) strategies. To maintain consistency of the calculus, we separate the (co-)data types into different *kinds* that denote different strategies, so that well-kinded programs are consistent. When considering only one strategy, this degenerates into the previous untyped equational theory, and for two strategies the approach is similar to Zeilberger's [14] "bi-typed" system, except generalized to also work with any number of additional strategies like call-by-need (or its dual).

Our contributions are: (1) We develop a parametric equational theory for the sequent calculus that may be instantiated by various strategies. We express the essence of a strategy by what may be substituted for a (co-)variable. In other words, a strategy is identified by a choice of values and co-values. (2) We enrich the sequent calculus with user-defined data and co-data types, whose behavior are defined exclusively in terms of β and η principles that do not refer to the chosen strategy. These two principles provide the basis for all user-defined (co-)data types, and may be used to derive other properties like Wadler's ς rules [11]. (3) We give call-by-value and call-by-name strategies for the equational theory that are sound and complete with respect to known CPS transformations [3] and their extension with user-defined (co-)data types. (4) We generalize the known duality of call-by-value and call-by-name evaluation in the sequent calculus [3,11] to be parametric over evaluation strategies and types, giving a mechanical procedure for generating the dual language for any choice of connectives and evaluation strategy, as expressed in the parametric calculus. (5) We exhibit that the parametric theory supports more intricate notions of evaluation strategy by instantiating it with a call-by-need strategy and generating its dual. (6) We illustrate how to compose two or more primitive strategies, such as call-by-value, call-by-name, and call-by-need, into a single composite strategy, so that a program may selectively choose and switch between several evaluation strategies at run-time.

2 Introduction to the sequent calculus

When implementing an evaluator for the λ-calculus, it becomes necessary to find the next reduction, or step, to perform in a term. Searching for the next reduction is not always trivial, since it may be buried deep inside the syntax of the term. For instance, consider the syntax tree for the term $(((\lambda x.M)\ N_1)\ N_2)\ N_3$ as shown in Figure 1(a), where the name α is a placeholder for the rest of the surrounding context. The next step is to call the function $(\lambda x.M)$ with the argument N_1, but



(a) λ-calculus (b) $\overline{\lambda}\mu\tilde{\mu}$-calculus

Fig. 1. Re-association of the abstract-syntax tree for function calls.

the term for this function call is at the bottom of the tree, and to reach it we need to search past the function calls with N_3 and N_2 as arguments. As an alternate representation of the same program, we can re-associate the syntax tree so that the next step to perform is located at the top of the tree, as shown in Figure 1(b). Imagine that we take hold of the edge connecting the function to its call and drag it upward so that the rest of the tree hangs off both sides of the edge, turning the context inside out. Syntactically, this amounts to converting the evaluation context into a term in its own right, *i.e.,* a *co-term*. Written out sequentially using Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [3], the re-associated program is the *command* $\langle \lambda x.M \| N_1 \cdot N_2 \cdot N_3 \cdot \alpha \rangle$, where $\cdot$ builds a call-stack and associates to the right. From the perspective of the Curry-Howard correspondence, this change in orientation in the syntax of programming languages corresponds with a similar change in the structure of proofs. Just as the λ-calculus corresponds with natural deduction, the $\overline{\lambda}\mu\tilde{\mu}$-calculus corresponds with the sequent calculus.

Fundamentally, the $\overline{\lambda}\mu\tilde{\mu}$-calculus describes computation as the interaction between a term and a co-term. For example, if we evaluate the program in Figure 1(b) according to a call-by-name strategy, where a function call is performed without evaluating the argument, we get the reduction

$$\langle \lambda x.M \| N_1 \cdot N_2 \cdot N_3 \cdot \alpha \rangle \twoheadrightarrow \langle M\,\{N_1/x\} \| N_2 \cdot N_3 \cdot \alpha \rangle$$

where we take the first argument, N_1, in the co-term and substitute it for x in the body of the function. Afterward, we evaluate the interaction between $M\,\{N_1/x\}$ and the remaining co-term. Alternatively, we may want to consider a call-by-value strategy, where we evaluate N_1 before calling the function. As a way to keep reduction at the top of the syntax tree, a $\tilde{\mu}$-abstraction can give a name to N_1. The co-term $\tilde{\mu}x.\langle M \| N_2 \cdot N_3 \cdot \alpha \rangle$ should be read as $\mathbf{let}\, x = \Box\, \mathbf{in}\, (M\ N_2\ N_3)$ in the context α. Therefore, we can make the call-by-value reduction

$$\langle \lambda x.M \| N_1 \cdot N_2 \cdot N_3 \cdot \alpha \rangle \twoheadrightarrow \langle N_1 \| \tilde{\mu}x.\langle M \| N_2 \cdot N_3 \cdot \alpha \rangle \rangle$$

with the understanding that we must first fully evaluate N_1 to a value before substituting it for x in $\langle M \| N_2 \cdot N_3 \cdot \alpha \rangle$. In addition to $\tilde{\mu}$-abstraction, we have the dual notion of μ-abstraction that allows a term to name its co-term. Therefore, we can close off the command by introducing α, giving us the term $\mu\alpha.\langle (\lambda x.M) \| N_1 \cdot N_2 \cdot N_3 \cdot \alpha \rangle$.

The $\overline{\lambda}\mu\tilde{\mu}$-calculus takes implication (functions) as its only logical connective (type constructor). However, we want to explore a variety of other connectives in the sequent calculus. Furthermore, once we have a method for declaring new type constructors, functions just become another instance of a user-defined type. For this reason, we temporarily forgo functions and more closely examine the core language of substitution.

3 The parametric $\mu\tilde{\mu}$ core

We now consider the μ and $\tilde{\mu}$-abstractions which lie at the heart of the $\overline{\lambda}\mu\tilde{\mu}$-calculus. More specifically, programs in the $\mu\tilde{\mu}$-calculus are defined as follows:

$$c \in Command ::= \langle v \| e \rangle \quad v \in Term ::= x \mid \mu\alpha.c \quad e \in CoTerm ::= \alpha \mid \tilde{\mu}x.c$$

The μ and $\tilde{\mu}$-abstractions, $\mu\alpha.c$ and $\tilde{\mu}x.c$ respectively, embody the primitive variable binding structure of the language, giving a name to a (co-)term in an underlying command. It follows that during evaluation, these abstractions implement a notion of substitution. The μ axiom gives control to the term (producer) by substituting the co-term for a co-variable, whereas the $\tilde{\mu}$ gives control to the co-term (consumer) by performing the opposite substitution:

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

As is, this theory is not consistent, as shown by the fact that the μ and $\tilde{\mu}$ axioms fight for control in the command $\langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle$[1]. To restore consistency, we can give priority to one axiom over the other [3]:

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

In lieu of considering two (or more) different theories that place restrictions where necessary, we instead give a single parametric equational theory that does not assume a particular evaluation strategy *a priori*. The theory is parameterized by a choice of *strategy*, $\mathcal{S}$, which is defined as a set of *values* and *co-values* that are subsets of terms and co-terms, respectively. The axioms for the parametric core $\mu\tilde{\mu}_{\mathcal{S}}$-calculus are given in Figure 2, where the meta-variables V and E range over the set of values and co-values given by $\mathcal{S}$, respectively. In addition to the substitution axioms, $\tilde{\mu}_V$ and μ_E, we also have extensionality axioms, η_μ and $\eta_{\tilde{\mu}}$, that eliminate trivial μ- and $\tilde{\mu}$-abstractions. Note that all equations follow the usual restrictions to avoid capture of static variables. For instance, α is not a free variable of v in the η_μ axiom of Figure 2.

Since the μ_E and $\tilde{\mu}_V$ axioms are restricted by the strategy, carefully chosen combinations of values and co-values may avoid the fundamental inconsistency of the calculus. The simplest consistent choice of (co-)values that we can make is to always exclude either μ or $\tilde{\mu}$-abstractions, as shown in Figure 3. We can

[1] If $\alpha \notin FV(c)$ and $x \notin FV(c')$ then $\langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle = c, c'$, equating arbitrary c and c'.

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

Fig. 2. The parametric equational theory $\mu\tilde{\mu}_{\mathcal{S}}$.

$$V \in Value_{\mathcal{N}} ::= v \quad E \in CoValue_{\mathcal{N}} ::= \alpha \quad V \in Value_{\mathcal{V}} ::= x \quad E \in CoValue_{\mathcal{V}} ::= e$$

Fig. 3. Call-by-name ($\mathcal{N}$) and call-by-value ($\mathcal{V}$) strategies of $\mu\tilde{\mu}_{\mathcal{S}}$

then form a core call-by-name evaluation strategy, $\mathcal{N}$, by letting every term be a value, and restricting co-values to just co-variables. Dually, we have a core call-by-value evaluation strategy, $\mathcal{V}$, by letting every co-term be a co-value, and restricting values to just variables. To disambiguate the different instances of the parametric equational theory $\mu\tilde{\mu}_{\mathcal{S}}$, we write $\mu\tilde{\mu}_{\mathcal{N}} \vdash c = c'$ and $\mu\tilde{\mu}_{\mathcal{V}} \vdash c = c'$ to mean that c and c' are equated by the $\mathcal{N}$ and $\mathcal{V}$ instances of the parametric theory, respectively. Notice that in both the $\mathcal{N}$ and $\mathcal{V}$ strategies, (co-)variables are considered co-values, which is a condition we always assume to hold when speaking of strategies in general. Moreover, the $\mu\tilde{\mu}_{\mathcal{S}}$ equational theory is closed under substitution of (co-)values for (co-)variables.

Finally, we can also give a *continuation-passing style* (CPS) transformation that maps sequent calculus programs to the λ-calculus. The CPS transformation can be used as a reference point for reasoning about the correctness of the equational theory through the usual β and η axioms in the resulting λ-calculus term. In Figure 4, we recount the call-by-name and call-by-value CPS transformations given in [3] for the core calculus, denoted $[\![\_]\!]^{\mathcal{N}}$ and $[\![\_]\!]^{\mathcal{V}}$. The $\mu\tilde{\mu}_{\mathcal{N}}$ and $\mu\tilde{\mu}_{\mathcal{V}}$ equational theories are sound and complete with respect to $\beta\eta$ equality of the λ-calculus terms resulting from the $[\![\_]\!]^{\mathcal{N}}$ and $[\![\_]\!]^{\mathcal{V}}$ transformations, respectively.

4 Functions

Having first laid out the core $\mu\tilde{\mu}_{\mathcal{S}}$-calculus, we consider the behavior of functions in more detail. Using the same notation as the $\overline{\lambda}\mu\tilde{\mu}$-calculus [3], we extend the core $\mu\tilde{\mu}_{\mathcal{S}}$-calculus with the following syntax, giving us the $\mu\tilde{\mu}\vec{\mathcal{S}}$-calculus:

$$v \in Term ::= \ldots \mid \lambda x.v \qquad e \in CoTerm ::= \ldots \mid v \cdot e$$

Functions are expressed as λ-abstraction terms ($\lambda x.v$), the same as in the λ-calculus. A function call, on the other hand, is represented by the co-term $v \cdot e$, where v stands for the function's argument and e for the calling context, which we first saw in Section 2. Additionally, we may extend our core call-by-name and call-by-value strategies from Figure 3 to account for functions, as shown in Figure 5. In the call-by-value strategy $\mathcal{V}$, we admit λ-abstractions as values and continue to let every co-term be a co-value. In the call-by-name strategy $\mathcal{N}$, we continue to let every term be a value, and admit the co-term $v \cdot E$ as a co-value, representing a λ-calculus context of the form $E[\Box\ v]$.

$$[\![\langle v \| e \rangle]\!]^{\mathcal{N}} \triangleq [\![e]\!]^{\mathcal{N}}\ [\![v]\!]^{\mathcal{N}} \qquad [\![\langle v \| e \rangle]\!]^{\mathcal{V}} \triangleq [\![v]\!]^{\mathcal{V}}\ [\![e]\!]^{\mathcal{V}}$$

$$[\![\alpha]\!]^{\mathcal{N}} \triangleq \lambda x.x\ \alpha \qquad [\![x]\!]^{\mathcal{N}} \triangleq x \qquad [\![x]\!]^{\mathcal{V}} \triangleq \lambda\alpha.\alpha\ x \qquad [\![\alpha]\!]^{\mathcal{V}} \triangleq \alpha$$

$$[\![\tilde{\mu}x.c]\!]^{\mathcal{N}} \triangleq \lambda x.[\![c]\!]^{\mathcal{N}} \quad [\![\mu\alpha.c]\!]^{\mathcal{N}} \triangleq \lambda\alpha.[\![c]\!]^{\mathcal{N}} \quad [\![\mu\alpha.c]\!]^{\mathcal{V}} \triangleq \lambda\alpha.[\![c]\!]^{\mathcal{V}} \quad [\![\tilde{\mu}x.c]\!]^{\mathcal{V}} \triangleq \lambda x.[\![c]\!]^{\mathcal{V}}$$

Fig. 4. Call-by-name ($\mathcal{N}$) and call-by-value ($\mathcal{V}$) CPS transformations of $\mu\tilde{\mu}_{\mathcal{S}}$.

$$V \in Value_{\mathcal{N}} ::= v \qquad V \in Value_{\mathcal{V}} ::= x \mid \lambda x.v$$

$$E \in CoValue_{\mathcal{N}} ::= \alpha \mid v \cdot E \qquad E \in CoValue_{\mathcal{V}} ::= e$$

Fig. 5. Call-by-name ($\mathcal{N}$) and call-by-value ($\mathcal{V}$) strategies of $\mu\tilde{\mu}^{\rightarrow}_{\mathcal{S}}$.

Now, we need to determine which axioms to add to our equational theory in order to give a complete account for the run-time behavior of functions. It is obvious we need an axiom for β reduction, as given in [3] for $\overline{\lambda}\mu\tilde{\mu}$, since that is the primary computational rule for functions. In addition, we also consider an axiom for η equality, giving functions a notion of extensionality similar to the λ-calculus. We therefore extend the core $\mu\tilde{\mu}_{\mathcal{S}}$ equational theory from Figure 2 with the two rules for functions in Figure 6 to obtain the $\mu\tilde{\mu}^{\rightarrow}_{\mathcal{S}}$-calculus. Notice that unlike in [3,11], these rules define the behavior of functions independently of the strategy since the $\beta^{\rightarrow}$ and $\eta^{\rightarrow}$ axioms do not reference V or E — the evaluation strategy is implemented by the core $\mu\tilde{\mu}_{\mathcal{S}}$-calculus alone.

We should ask ourselves if these rules make sense computationally, so that a consistent strategy for the core calculus is still consistent when extended with functions. The $\beta^{\rightarrow}$ axiom is applicable to any command between a λ-abstraction and a call, and dissolves the function call into a $\tilde{\mu}$ binding, thereby relying on the consistency of the strategy in the core $\mu\tilde{\mu}_{\mathcal{S}}$-calculus. For example, with the call-by-value strategy $\mathcal{V}$, the $\tilde{\mu}$-abstraction bears the responsibility of ensuring that the argument v' to a function call is a value before substituting it into the body of the function — if v' is the non-value $\mu\alpha.c$ then it gets to go first by means of the μ_E rule. On the other hand, the $\eta^{\rightarrow}$ axiom is restricted to apply only to variables. Intuitively, the $\eta^{\rightarrow}$ axiom states that an unknown function is indistinguishable from a λ-abstraction. Recall that the core $\mu\tilde{\mu}_{\mathcal{S}}$ theory is closed under substitution of values for variables, so the usual $\eta^{\rightarrow}$ axiom that applies to values is derivable within the equational theory. This restriction is crucial for preserving consistency of certain strategies, like the $\mathcal{V}$ strategy, and comes out for free from the meaning of $\tilde{\mu}$-abstractions in the core calculus.

We should also ask ourselves if these rules are complete enough to describe the other behavioral properties of functions. For instance, previous reduction systems for the sequent calculus [11,8] include a family of ς rules that lift sub-computations to the top of a command. For implication, this takes the form of axioms that lift out the sub-expressions of a function call:

$$(\varsigma^{\rightarrow}_{x}) \quad v \cdot e = \tilde{\mu}f.\langle v \| \tilde{\mu}x.\langle f \| x \cdot e \rangle \rangle \qquad (\varsigma^{\rightarrow}_{\alpha}) \quad V \cdot e = \tilde{\mu}f.\langle \mu\alpha.\langle f \| V \cdot \alpha \rangle \| e \rangle$$

The ς rules are necessary for making progress with some programs. For example, call-by-value evaluation of the command $\langle z \| \mu\alpha.c \cdot e \rangle$ cannot proceed by β reduc-

$$(\beta^{\rightarrow}) \qquad \langle \lambda x.v \| v' \cdot e \rangle = \langle v' \| \tilde{\mu} x.\langle v \| e \rangle \rangle \qquad (\eta^{\rightarrow}) \qquad \lambda x.\mu\alpha.\langle z \| x \cdot \alpha \rangle = z$$

Fig. 6. The β and η axioms of the $\mu\tilde{\mu}^{\rightarrow}_{\mathcal{S}}$ equational theory.

$$\begin{aligned} [\![\lambda x.v]\!]^{\mathcal{N}} &\triangleq \lambda(x,\beta).[\![v]\!]^{\mathcal{N}}\beta & [\![\lambda x.v]\!]^{\mathcal{V}} &\triangleq \lambda\alpha.\alpha\ \lambda(x,\beta).[\![v]\!]^{\mathcal{V}}\beta \\ [\![v \cdot e]\!]^{\mathcal{N}} &\triangleq \lambda x.[\![e]\!]^{\mathcal{N}}\lambda\beta.x\ ([\![v]\!]^{\mathcal{N}},\beta) & [\![v \cdot e]\!]^{\mathcal{V}} &\triangleq \lambda x.[\![v]\!]^{\mathcal{V}}\lambda y.x\ (y,[\![e]\!]^{\mathcal{V}}) \end{aligned}$$

Fig. 7. Call-by-name ($\mathcal{N}$) and call-by-value ($\mathcal{V}$) CPS transformations of functions.

tion, and since the non-value argument $\mu\alpha.c$ is buried inside of a function call, it needs to be lifted out for it to take control so that evaluation can continue. A course-grained version of the lifting axiom, which lifts out both parts of a function call

$$(\varsigma^{\rightarrow}) \qquad v \cdot e = \tilde{\mu} f.\langle v \| \tilde{\mu} x.\langle \tilde{\mu}\alpha.\langle f \| x \cdot \alpha \rangle \| e \rangle \rangle$$

is easily derived from the $\eta_{\tilde{\mu}}$, $\eta^{\rightarrow}$, and $\beta^{\rightarrow}$ axioms as follows:

$$v \cdot e =_{\eta_{\tilde{\mu}}} \tilde{\mu} f.\langle f \| v \cdot e \rangle =_{\eta^{\rightarrow}} \tilde{\mu} f.\langle \lambda x.\mu\alpha.\langle f \| x \cdot \alpha \rangle \| v \cdot e \rangle =_{\beta^{\rightarrow}} \tilde{\mu} f.\langle v \| \tilde{\mu} x.\langle \mu\alpha.\langle f \| x \cdot \alpha \rangle \| e \rangle \rangle$$

Furthermore, the $\varsigma^{\rightarrow}$ axiom can be broken down into the more atomic rules within the existing equational theory. The derivation of $\varsigma^{\rightarrow}_{\alpha}$ from $\varsigma^{\rightarrow}$ is a consequence of the $\tilde{\mu}_V$ axiom, and we can derive the $\varsigma^{\rightarrow}_{x}$ axiom as follows:

$$\begin{aligned} v \cdot e &=_{\varsigma^{\rightarrow}} \tilde{\mu} f.\langle v \| \tilde{\mu} x.\langle \mu\alpha.\langle f \| x \cdot \alpha \rangle \| e \rangle \rangle \\ &=_{\tilde{\mu}_V} \tilde{\mu} f.\langle v \| \tilde{\mu} x.\langle f \| \tilde{\mu} f'.\langle x \| \tilde{\mu} x'.\langle \mu\alpha.\langle f' \| x' \cdot \alpha \rangle \| e \rangle \rangle \rangle \rangle =_{\varsigma^{\rightarrow}} \tilde{\mu} f.\langle v \| \tilde{\mu} x.\langle f \| x \cdot e \rangle \rangle \end{aligned}$$

Therefore, the combination of $\beta^{\rightarrow}$ and $\eta^{\rightarrow}$ axioms is powerful enough in the parametric equational theory $\mu\tilde{\mu}^{\rightarrow}_{\mathcal{S}}$ to express other known behavioral properties of functions that are needed for certain strategies.

We can achieve a more concrete sense of completeness for the specific cases of call-by-name and call-by-value functions by extending our core CPS transformations for the $\mathcal{N}$ and $\mathcal{V}$ strategies, as in $\overline{\lambda}\mu\tilde{\mu}$ [3] and shown in Figure 7, with clauses that handle function abstractions and calls. The $\mu\tilde{\mu}^{\rightarrow}_{\mathcal{N}}$ and $\mu\tilde{\mu}^{\rightarrow}_{\mathcal{V}}$ equational theories are sound and complete with respect to the $[\![\_]\!]^{\mathcal{N}}$ and $[\![\_]\!]^{\mathcal{V}}$ CPS transformations, respectively. This was previously known to hold for two separate and disjoint subsets of $\overline{\lambda}\mu\tilde{\mu}$ [3], but we now show that the correspondence holds for the full $\mu\tilde{\mu}^{\rightarrow}_{\mathcal{N}}$- and $\mu\tilde{\mu}^{\rightarrow}_{\mathcal{V}}$-calculi using the equational theories given by the strategies presented here.

Theorem 1. *$\mu\tilde{\mu}^{\rightarrow}_{\mathcal{S}} \vdash c = c'$ if and only if $\beta\eta \vdash [\![c]\!]^{\mathcal{S}} = [\![c']\!]^{\mathcal{S}}$, for $\mathcal{S} = \mathcal{V}$ or $\mathcal{N}$.*

Therefore, it turns out that the combination of the $\beta^{\rightarrow}$ and $\eta^{\rightarrow}$ axioms alone really do give a complete account of functions in the sequent calculus. Furthermore, both of these axioms do not reference the strategy: all of the details regarding order of evaluation has been taken care of by the core $\mu\tilde{\mu}_{\mathcal{S}}$-calculus.

Remark 1. Now that we have a logical connective to work with, we can compare our use of strategies for determining evaluation order with the use of types in polarized logic. Take the usual ambiguous command, $\langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle$, in which both sides appear to be fighting for control, and let's assume that the term and co-term belong to the type $A \to B$. One way to resolve the conflict is to assume that the $\eta^{\to}$ rule, corresponding to the reversibility of implication introduction in a proof or the universal property of exponentials in a category, is as strong as possible. Under this assumption, we can use the $\eta^{\to}$ rule to expand any term of type $A \to B$ into a λ-abstraction. Therefore, the ambiguous command is actually equivalent to the unambiguous command $\langle \lambda y.v \| \tilde{\mu}x.c' \rangle$, where $v = \mu\beta.\langle \mu\alpha.c \| y \cdot \beta \rangle$, and the conflict has been resolved in favor of the consumer. This goes to show that under the polarized view of types, in which the η rules are taken to be as strong as possible, the type of the active (co-)terms in a command can be used to determine evaluation order. Since the η rules for negative types like $A \to B$ apply to terms, then every term must be equivalent to a value, leading to a call-by-name interpretation. Dually, the η rules for positive types apply to co-terms, so every co-term must be a co-value, giving us a call-by-value interpretation. In contrast, the strategy based interpretation allows the user of the equational theory to choose what is considered a (co-)value, and the logical η rules are weakened so that they are consistent with the choice of strategy.

5 Basic data and co-data structures

So far, our approach has been to characterize the behavior of functions in terms of β and η axioms alone, giving us a complete axiomatization for functions in the sequent calculus. All other details relevant to computation, such as when to lift out sub-computations in a function call, are derived from the primitive β and η principles. Furthermore, the β and η rules did not directly reference the strategy, but instead the meaning of the strategy is entirely defined by the core $\mu\tilde{\mu}_{\mathcal{S}}$ calculus. To demonstrate the general applicability of this approach, and to build toward a more complete language, we should also account for pairs and disjoint unions, giving us a notion of products and sums in the sequent calculus. As a test to see if our formulation of the β and η axioms are sufficient, we will derive similar lifting rules, ς, as those described in Section 4 for functions.

We begin by considering sums ($\oplus$) in the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus. As per the usual approach in functional programming languages (based on natural deduction style), terms are injected into the sum as $\iota_1(v)$ or $\iota_2(v)$, and later analyzed by cases in the form $\mathbf{case}\, v\, \mathbf{of}\, \iota_1(x) \Rightarrow v_1 | \iota_2(y) \Rightarrow v_2$. In the sequent setting, we can keep the same terms, and reify the context for case analysis into the co-term $\tilde{\mu}[\iota_1(x).c_1 | \iota_2(y).c_2]$. Our goal now is to characterize the dynamic behavior of sums in terms of β and η axioms. Performing β reduction is implemented by a straightforward case analysis, matching the tag of the term with the appropriate branch of the co-term. For the η rule, we want to recognize a trivial case analysis that rebuilds the sum exactly as it was. Therefore, to extend the core

$\mu\tilde{\mu}_{\mathcal{S}}$-calculus with sums, we include the following two axioms:

$$(\beta^{\oplus}) \qquad \langle \iota_i(v) \| \tilde{\mu}[\iota_1(x).c_1 | \iota_2(x).c_2] \rangle = \langle v \| \tilde{\mu}x.c_i \rangle$$
$$(\eta^{\oplus}) \qquad \tilde{\mu}[\iota_1(x).\langle \iota_1(x) \| \alpha \rangle | \iota_2(y).\langle \iota_2(y) \| \alpha \rangle] = \alpha$$

Notice in particular that under call-by-value evaluation, the β axiom is applicable even when $\iota_i(v)$ is not a value, which is not directly allowed in Wadler's [11] call-by-value sequent calculus but is sound with respect to the CPS transformation. As with the β rule for functions, we are relying on the fact that a $\tilde{\mu}$-abstraction establishes the correct evaluation order, so that the underlying term will only be substituted if it is a value. Additionally, substitution of a co-value for α means that the η axiom for sums is also applicable to co-values. Because of the ability to substitute a co-value for a co-variable, we end up with a stronger η axiom for sums than we may have otherwise considered in natural deduction, corresponding to $(\mathbf{case}\, v\, \mathbf{of}\, \iota_1(x) \Rightarrow E[\iota_1(x)] | \iota_2(x) \Rightarrow E[\iota_2(x)]) = E[v]$ where E is an evaluation context of the chosen strategy. Restricting the η axiom to co-variables captures the fact that languages which impose restrictions on co-values have a correspondingly restricted notion of sums, as observed by Filinski for call-by-name languages [5]. To test that this combination of β and η axioms completely defines the behavior of sums, we derive Wadler's [12] lifting rule: $(\varsigma^{\oplus}) \quad \iota_i(v) = \mu\alpha.\langle v \| \tilde{\mu}x.\langle \iota_i(x) \| \alpha \rangle \rangle$

$$\iota_i(v) =_{\eta_\mu, \eta^{\oplus}} \mu\alpha.\langle \iota_i(v) \| \tilde{\mu}[\iota_1(x).\langle \iota_1(x) \| \alpha \rangle | \iota_2(x).\langle \iota_2(x) \| \alpha \rangle] \rangle =_{\beta^{\oplus}} \mu\alpha.\langle v \| \tilde{\mu}x.\langle \iota_i(x) \| \alpha \rangle \rangle$$

The fact that we have unrestricted β reduction for sums is crucial for deriving the $\varsigma^{\oplus}$ axiom. If we were only allowed to work with values, then the second step of the derivation would not be possible.

Next, we would like to formulate products ($\otimes$) that correspond to eager pairs in the call-by-value setting. Constructing a pair can be given straightforwardly as (v_1, v_2), following natural deduction style. Suppose now that we choose to define the co-terms as the projections $\pi_1[e]$ and $\pi_2[e]$, to correspond with the natural deduction terms $\pi_1(v)$ and $\pi_2(v)$. In order to implement eager pairs using this formulation, we would be forced to restrict β reduction to commands of the form $\langle (V_1, V_2) \| \pi_i[e] \rangle$, since we can only project out of an eager product when both components are values. This restriction on the β axiom makes it impossible to derive the appropriate lifting axioms for eager products, which means that the β and η axioms would be necessarily incomplete. The fundamental problem is that this formulation of pairs does not give us a $\tilde{\mu}$-abstraction to rely on for evaluating the sub-terms, forcing us to infect the β rule with details about evaluation order. Instead, we define the co-term as a case abstraction, $\tilde{\mu}(x, y).c$, which corresponds to case analysis on the structure of a pair in natural deduction, $\mathbf{case}\, v\, \mathbf{of}\, (x, y) \Rightarrow v'$. As before, β reduction decomposes the structure, and the η axiom recognizes a trivial case abstraction that immediately rebuilds the pair:

$$(\beta^{\otimes}) \quad \langle (v_1, v_2) \| \tilde{\mu}(x, y).c \rangle = \langle v_1 \| \tilde{\mu}x.\langle v_2 \| \tilde{\mu}y.c \rangle \rangle \quad (\eta^{\otimes}) \quad \tilde{\mu}(x, y).\langle (x, y) \| \alpha \rangle = \alpha$$

Notice that the β axiom is strong enough to break apart any pair (v_1, v_2) without throwing anything away, allowing us to still evaluate the two sub-terms eagerly

afterward with the $\tilde{\mu}$-abstractions generated by the β rule. For instance, in the call-by-value strategy $\mathcal{V}$, the command $\langle (V, \mu\_.c') \| \tilde{\mu}(x, \_).c \rangle$ rightly reduces to c'. Additionally, because the β axiom breaks apart a pair containing (potentially) non-values, it must give an order to the bindings of the elements, thereby determining an order of evaluation between them. In the β rule presented here, we (arbitrarily) give priority to the first component of the pair. We can now pass our test by deriving a lifting rule for products that pulls out the two components so that they may be evaluated: $(\varsigma^{\otimes}) \quad (v_1, v_2) = \mu\alpha.\langle v_1 \| \tilde{\mu}x.\langle v_2 \| \tilde{\mu}y.\langle (x, y) \| \alpha \rangle \rangle \rangle$

$$(v_1, v_2) =_{\eta_\mu, \eta^{\otimes}} \mu\alpha.\langle (v_1, v_2) \| \tilde{\mu}(x, y).\langle (x, y) \| \alpha \rangle \rangle =_{\beta^{\otimes}} \mu\alpha.\langle v_1 \| \tilde{\mu}x.\langle v_2 \| \tilde{\mu}y.\langle (x, y) \| \alpha \rangle \rangle \rangle$$

As with functions, we derive Wadler's [12] more atomic rules that lift out one term at a time:

$$(\varsigma_x^{\otimes}) \quad (v_1, v_2) = \mu\alpha.\langle v_1 \| \tilde{\mu}x.\langle (x, v_2) \| \alpha \rangle \rangle \quad (\varsigma_x^{\otimes}) \quad (V_1, v_2) = \mu\alpha.\langle v_2 \| \tilde{\mu}y.\langle (V_1, y) \| \alpha \rangle \rangle$$

The symmetry of the sequent calculus points out a dual formulation of pairs and sums. This corresponds to the two forms of conjunction and disjunction in Girard's linear logic [6] and polarized logic [13,4]. Taking the mirror image of sums ($\oplus$) gives a formulation of products (&) using projection as primitive that computes either the first or the second component on demand. The mirror image of products ($\otimes$) gives us a "classical" disjunction ($\parr$), resulting in a lazier sum which only evaluates the term as it is needed, once both branches of its co-term have been reduced to co-values. The syntax and axioms for these connectives are exactly dual to those given above. The & connective has terms of the form $\mu(\pi_1[\alpha].c_1 | \pi_2[\beta].c_2)$ and the co-terms $\pi_1[e]$ and $\pi_2[e]$, and the $\parr$ connective has the term $\mu[\alpha, \beta].c$ and the co-term $[e_1, e_2]$. For example, $\mu(\pi_1[\alpha].\langle 1 \| \alpha \rangle | \pi_2[\beta].\langle 2 \| \beta \rangle)$ is a & product that immediately returns 1 or 2 when asked, and given a & product x, we may swap its responses by intercepting and reversing the messages it receives: $\mu(\pi_1[\alpha].\langle x \| \pi_2[\alpha] \rangle | \pi_2[\beta].\langle x \| \pi_1[\beta] \rangle)$. Additionally, a $\parr$ term may return a result to one of the two branches by responding to one of the provided co-variables, for example responding with 1 to the left branch is written $\mu[\alpha, \beta].\langle 1 \| \alpha \rangle$. Intuitively, $\oplus$ and & express the concept of choice (the choice to produce either the first or second or the choice to ask for the first or second), whereas $\otimes$ and $\parr$ are about an amalgamation of two sub-parts.

To finish off the development, we also extend our call-by-value and call-by-name strategies to account for the new (co-)terms. We extend the sets of $\mathcal{V}$ values and $\mathcal{N}$ co-values as

$$V \in Value_{\mathcal{V}} ::= \ldots \mid (V, V') \mid \iota_i(V) \mid \mu(\pi_1[\alpha].c | \pi_2[\beta].c') \mid \mu[\alpha, \beta].c$$
$$E \in CoValue_{\mathcal{N}} ::= \ldots \mid \tilde{\mu}(x, y).c \mid \tilde{\mu}[\iota_1(x).c | \iota_2(y).c'] \mid \pi_i[E] \mid [E, E']$$

and continue to accept every co-term as a $\mathcal{V}$ co-value and every term as a $\mathcal{N}$ value. Notice in particular that the $\mathcal{V}$ strategy has a notion of eager and non-eager pairs: the concrete $\otimes$ term, (v, v'), will eagerly evaluate its sub-terms before becoming a value, whereas the abstract & term, $\mu(\pi_1[\alpha].c | \pi_2[\beta].c')$, is a value that is waiting for a message before running one of its sub-commands. The meanings

of functions and & are similar in call-by-value, where we eagerly evaluate a term down to an abstraction and then stop. On the other hand, the $\mathcal{N}$ strategy implements the idea of a strict and non-strict sum: the $\oplus$ case abstraction is a co-value that forces evaluation of its term, whereas the $\parr$ co-structure only forces evaluation of its term when both branches are co-values, so that they are strict in their input. This fundamental difference of the two views on disjunction has been previously observed by Selinger [10], who pointed out that in call-by-name, the two forms of disjunction cannot be isomorphic to one another. We also have the dual property, that there are two fundamentally different forms of products in call-by-value: a concrete pair and an abstract pair.

6 User-defined data and co-data types

By this point, we have arrived at a common pattern for adding basic (co-)data types to the core $\mu\tilde{\mu}_{\mathcal{S}}$-calculus, which we will now generalize to user-defined types, similar to Herbelin's notion of generalized connectives [7]. We will take the data or co-data nature of a type constructor as a fundamental ingredient to its definition, therefore allowing the user to declare new data types (with concrete terms) and co-data types (with abstract terms). These are two dual ways of approaching data structures in programming languages: data corresponds to ordinary data types in functional languages like ML, whereas co-data is more akin to an interface for abstract objects that defines a fixed set of allowable observations or messages. The utility of definition by observations has been previously shown for infinite structures [1]. We present (co-)data declarations in the style of a statically typed language, like Haskell. However, since we are focused on an equational theory in an untyped setting, we use the declarations as a way to extend the language with new syntactic forms for structures and abstractions and to extend the theory with rules defining their operational meaning.

6.1 Defining basic data and co-data types

We first approach user-defined (co-)data types by example, and observe how the basic type constructors we have considered so far fit within the same general framework. To express the declarations in their full generality, we use a richer notation than that provided for ordinary algebraic data types in ML. Therefore, consider how the syntax of GADT declarations in Haskell can be applied to ordinary algebraic data types. For instance, the basic Either and Both (a tuple of two components) type constructors in Haskell are declared using GADT notation as follows:

$$\begin{array}{l}\textbf{data}\ \mathsf{Either}\ A\ B\ \textbf{where}\\ \qquad \mathsf{Left} : A \to \mathsf{Either}\ A\ B\\ \qquad \mathsf{Right} : B \to \mathsf{Either}\ A\ B\end{array} \qquad \begin{array}{l}\textbf{data}\ \mathsf{Both}\ A\ B\ \textbf{where}\\ \qquad \mathsf{Pair} : A \to B \to \mathsf{Both}\ A\ B\end{array}$$

The declaration of $\mathsf{Either}\ A\ B$ corresponds with the sequent declaration of $A \oplus B$ in Figure 8, both of which introduce a data type with two constructors:

data $A \oplus B$ **where**	**codata** $A \,\&\, B$ **where**
$\iota_1 : A \vdash A \oplus B \mid$	$\pi_1 : \mid A \,\&\, B \vdash A$
$\iota_2 : B \vdash A \oplus B \mid$	$\pi_2 : \mid A \,\&\, B \vdash B$
data $A \otimes B$ **where**	**codata** $A \parr B$ **where**
$\mathsf{pair} : A, B \vdash A \otimes B \mid$	$\mathsf{split} : \mid A \parr B \vdash A, B$
data 1 **where**	**codata** $\bot$ **where**
$\mathsf{unit} : \vdash 1 \mid$	$\mathsf{tp} : \mid \bot \vdash$
data 0 **where**	**codata** $\top$ **where**
data $A - B$ **where**	**codata** $A \to B$ **where**
$\mathsf{uncall} : A \vdash A - B \mid B$	$\mathsf{call} : A \mid A \to B \vdash B$

Fig. 8. Declarations for basic (co-)data types.

one accepting an input of type A and the other an input of type B. However, the sequent declaration separates input from output with entailment, $\vdash$, rather than a function arrow, and explicitly distinguishes the result produced by the constructor as $A \oplus B \mid$. Similarly, the constructor Pair from the declaration of $\mathsf{Both}\ A\ B$ can be seen as a curried form of the constructor pair from $A \otimes B$. In addition, the data declarations of 1 and 0 in Figure 8 correspond to the usual unit and empty types in functional programming languages.

However, the rest of the declarations in Figure 8 step outside the usual notion of data type in functional programming languages, and illustrate the various possibilities for defining new type constructors in the sequent calculus. The co-data declaration for $A \,\&\, B$ introduces a pair that is uniquely defined by their first and second projections, which consume the distinguished input written as $\mid A \,\&\, B$, rather than by a structure containing two elements. The declaration for $A \parr B$ demonstrates that a (co-)constructor in the sequent calculus may have multiple outputs. The co-data declarations for $\top$ and $\bot$ give a dual notion of the unit and empty types, respectively, where the unit is an abstract object with no possible observations, and the empty type has one observation that produces no output. We can also express implication, $A \to B$, and its dual, $A - B$, as user-defined types that make use of both input and output at the same time.

6.2 Defining new data and co-data types

Next, we consider how to introduce a new data type to the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus, in its full generality. A data type is defined by cases over a set of *constructors*, $\mathsf{K}_1, \ldots, \mathsf{K}_n$. The general form of declaration for the new data type $\mathsf{F}(\vec{X})$, where $\vec{X}$ are zero or more type variables[2], is given in Figure 9. The type variables may appear in any of the types $\vec{A}$ and $\vec{B}$, and each constructor has $\mathsf{F}(\vec{X})$ as the distinguished

[2] We write $\vec{X}_i^{\,i}$ to mean a sequence $X_1, \ldots, X_n$ of zero or more elements indexed by i. The index is left implicit when it is clear from context.

$$\begin{array}{ll}
\textbf{data}\ \mathsf{F}(\overrightarrow{X_j}^j)\ \textbf{where} & \textbf{codata}\ \mathsf{G}(\overrightarrow{X_j}^j)\ \textbf{where} \\
\quad \mathsf{K}_1 : \overrightarrow{A_{1j}}^j \vdash \mathsf{F}(\overrightarrow{X_j}^j) | \overrightarrow{B_{1j}}^j & \quad \mathsf{H}_1 : \overrightarrow{A_{1j}}^j \mid \mathsf{G}(\overrightarrow{X_j}^j) \vdash \overrightarrow{B_{1j}}^j \\
\quad \cdots & \quad \cdots \\
\quad \mathsf{K}_n : \overrightarrow{A_{nj}}^j \vdash \mathsf{F}(\overrightarrow{X_j}^j) | \overrightarrow{B_{nj}}^j & \quad \mathsf{H}_n : \overrightarrow{A_{nj}}^j \mid \mathsf{G}(\overrightarrow{X_j}^j) \vdash \overrightarrow{B_{nj}}^j
\end{array}$$

Fig. 9. The general forms of (co-)data declarations in $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$.

$$\begin{array}{ll}
(\beta^{\mathsf{F}}) & \langle \mathsf{K}_i(\overrightarrow{e_j}^j, \overrightarrow{v_j}^j) \| \tilde{\mu}[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_{ij}}^j, \overrightarrow{x_{ij}}^j).c_i}^i] \rangle = \langle \mu\overrightarrow{\alpha_{ij}}^j.\langle \overrightarrow{v_j}^j \| \tilde{\mu}\overrightarrow{x_{ij}}^j.c_i \rangle \| \overrightarrow{e_j}^j \rangle \\
(\eta^{\mathsf{F}}) & \tilde{\mu}[\overrightarrow{\mathsf{K}_i(\overrightarrow{\alpha_{ij}}^j, \overrightarrow{x_{ij}}^j).\langle \mathsf{K}_i(\overrightarrow{\alpha_{ij}}^j, \overrightarrow{x_{ij}}^j) \| \gamma \rangle}^i] = \gamma \\
(\beta^{\mathsf{G}}) & \langle \mu(\overrightarrow{\mathsf{H}_i[\overrightarrow{x_{ij}}^j, \overrightarrow{\alpha_{ij}}^j].c_i}^i) \| \mathsf{H}_i[\overrightarrow{v_j}^j, \overrightarrow{e_j}^j] \rangle = \langle \overrightarrow{v_j}^j \| \tilde{\mu}\overrightarrow{x_{ij}}^j.\langle \mu\overrightarrow{\alpha_{ij}}^j.c_i \| \overrightarrow{e_j}^j \rangle \rangle \\
(\eta^{\mathsf{G}}) & \mu(\overrightarrow{\mathsf{H}_i[\overrightarrow{x_{ij}}^j, \overrightarrow{\alpha_{ij}}^j].\langle z \| \mathsf{H}_i[\overrightarrow{x_{ij}}^j, \overrightarrow{\alpha_{ij}}^j] \rangle}^i) = z
\end{array}$$

Fig. 10. The β and η axioms for (co-)data of the $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$ equational theory.

output type on the right of the sequent. Syntactically, each constructor builds a new term not only from other terms, as per usual in a functional programming language, but also possibly from co-terms that represent reified contexts. The data declaration for $\mathsf{F}(\vec{X})$ introduces the family of data structures, $\mathsf{K}_i(\vec{e}, \vec{v})$, as new terms and the single case abstraction, $\tilde{\mu}[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c}]$, as the new co-term of that type. In addition to the syntax for the data type $\mathsf{F}(\vec{X})$, we also have two primitive axioms, β^{F} and η^{F} shown in Figure 10. Following the same pattern as pairs and sums, the axioms are strategy independent and rely on μ and $\tilde{\mu}$ in $\mu\tilde{\mu}_{\mathcal{S}}$ to manage evaluation order. Binding the sequence of terms $\vec{v} = v_1, v_2, \ldots, v_n$ to the sequence of variables $\vec{x} = x_1, x_2, \ldots, x_n$ is defined as

$$\langle \vec{v} \| \tilde{\mu}\vec{x}.c \rangle \triangleq \langle v_1 \| \tilde{\mu}x_1.\langle v_2 \| \tilde{\mu}x_2. \ldots \langle v_n \| \tilde{\mu}x_n.c \rangle \ldots \rangle \rangle$$

and analogously for binding a sequence of co-terms to co-variables. The β^{F} axiom performs case analysis by looking up the appropriate command to run based on the constructor and binding the sub (co-)terms of the structure by matching it with the appropriate pattern. The η^{F} axiom states that an unknown co-value γ is treated the same as a trivial case abstraction that re-constructs all matched structures and forwards them along to γ. As before, we have a family of axioms that lift out sub-(co-)terms in a data structure, which can be derived by following the same pattern shown in Section 5 for products and sums.

Introducing a new co-data type to the $\mu\tilde{\mu}_{\mathcal{S}}$-calculus follows the same general pattern, but with a twist. Instead of defining a co-data type $\mathsf{G}(\vec{X})$, by its constructors, which produce a data structure as output, it is defined by cases over its possible *co-constructors* which build concrete co-structures on the side of co-terms. The co-structures can be thought of as observations or messages that are sent to and analyzed by abstract terms of type $\mathsf{G}(\vec{X})$. It follows that

$$\left(\mathbf{data}\,\mathsf{F}(\vec{X})\,\mathbf{where}\,\overrightarrow{\mathsf{K} : \vec{A} \vdash \mathsf{F}(\vec{X})|\vec{B}}\right)^\circ \triangleq \mathbf{codata}\,\mathsf{F}(\vec{X})\,\mathbf{where}\,\overrightarrow{\mathsf{K} : \vec{B}|\,\mathsf{F}(\vec{X}) \vdash \vec{A}}$$

$$\left(\mathbf{codata}\,\mathsf{G}(\vec{X})\,\mathbf{where}\,\overrightarrow{\mathsf{H} : \vec{A}|\,\mathsf{G}(\vec{X}) \vdash \vec{B}}\right)^\circ \triangleq \mathbf{data}\,\mathsf{G}(\vec{X})\,\mathbf{where}\,\mathsf{H} : \vec{B} \vdash \mathsf{G}(\vec{X})|\vec{A}$$

$$\langle v \| e \rangle^\circ \triangleq \langle e^\circ \| v^\circ \rangle$$

$$(\mu\alpha.c)^\circ \triangleq \tilde{\mu}\alpha^\circ.c^\circ \quad (\mathsf{K}(\vec{e}, \vec{v}))^\circ \triangleq \mathsf{K}[\vec{e^\circ}, \vec{v^\circ}] \quad (\mu(\overrightarrow{\mathsf{H}[\vec{x}, \vec{\alpha}].c}))^\circ \triangleq \tilde{\mu}[\overrightarrow{\mathsf{H}(\vec{x^\circ}, \vec{\alpha^\circ}).c^\circ}]$$

$$(\tilde{\mu}x.c)^\circ \triangleq \mu x^\circ.c^\circ \quad (\mathsf{H}[\vec{v}, \vec{e}])^\circ \triangleq \mathsf{H}(\vec{v^\circ}, \vec{e^\circ}) \quad (\tilde{\mu}[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c}])^\circ \triangleq \mu(\overrightarrow{\mathsf{K}(\vec{\alpha^\circ}, \vec{x^\circ}).c^\circ})$$

Fig. 11. The duality operation for the $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus.

the co-data declaration mirrors the general form of a data declaration, as shown in Figure 9. Likewise, the syntax introduced for the co-data type $\mathsf{G}(\vec{X})$ is dual to the data form. We now have a family of co-data structures, $\mathsf{H}_i[\vec{v}, \vec{e}]$, as new co-terms, and a single new term, $\mu(\overrightarrow{\mathsf{H}[\vec{x}, \vec{\alpha}].c})$. This term is a co-case abstraction which responds to a co-data structure, *i.e.,* a message, by giving a command to perform for every possible case.

Following the exchanged roles between term and co-term, the primitive β^{G} and η^{G} axioms are mirror images of their counterparts for data types, as seen in Figure 10. The β^{G} axiom performs case analysis on a co-constructor, matching the co-structure to the given pattern and running the appropriate command given by the abstract term. The η^{G} axiom wraps an unknown value z in a co-case abstraction that just forwards every co-structure to that original value. We also have a family of axioms that lift out sub-(co-)terms in a co-data structure, which exactly mirror the derived lifting axioms for data types.

6.3 Duality and strategies for user-defined (co-)data types

We also extend the duality relationship of Curien and Herbelin [3] and Wadler [11] to be parametric over evaluation strategies and user-defined (co-)data types. The duality operation, given in Figure 11, transforms a program in the $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$-calculus into its dual in the $\mu\tilde{\mu}^{\mathcal{F}^\circ}_{\mathcal{S}^\circ}$-calculus. The duality operation flips the roles of a program, mapping terms into co-terms, co-terms into terms, and exchanges the two sides of a command. The dual of a set of declared (co-)data types, $\mathcal{F}$, is given by the dual of each (co-)data type declaration of $\mathcal{F}$. In addition, we can automatically generate the dual of a strategy, $\mathcal{S}^\circ$, by taking the point-wise dual of the values and co-values of $\mathcal{S}$. Notice that the double dual of a (co-)data declaration is identical to the original declaration. This gives us soundness and involution of duality that is parametric in evaluation strategy and (co-)data types.

Theorem 2. – *If* $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}} \vdash c = c'$ *then* $\mu\tilde{\mu}^{\mathcal{F}^\circ}_{\mathcal{S}^\circ} \vdash c^\circ = c'^\circ$.
– $c^{\circ\circ} \triangleq c$, $\mathcal{S}^{\circ\circ} \triangleq \mathcal{S}$, *and* $\mathcal{F}^{\circ\circ} \triangleq \mathcal{F}$.

It is worthwhile to pause over the statement of this theorem: for *every* strategy and collection of (co-)data types under which two commands are equated, the

$$V \in Value_{\mathcal{V}} ::= x \mid \mathsf{K}_i(\vec{e}, \vec{V}) \mid \mu(\overrightarrow{\mathsf{H}(\vec{x}, \vec{\alpha}).c}) \quad E \in CoValue_{\mathcal{V}} ::= e$$
$$V \in Value_{\mathcal{N}} ::= v \quad E \in CoValue_{\mathcal{N}} ::= \alpha \mid \tilde{\mu}[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c}] \mid \mathsf{H}_i[\vec{v}, \vec{E}]$$

Fig. 12. Call-by-value ($\mathcal{V}$) and call-by-name ($\mathcal{N}$) strategies of $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$.

$$[\![\mathsf{K}_i(\vec{e}, \vec{v})]\!]^{\mathcal{V}} = \lambda\alpha.[\![v_1]\!]^{\mathcal{V}}\lambda x_1 \ldots . [\![v_n]\!]^{\mathcal{V}}\lambda x_n.\alpha\ \ \mathsf{K}_i(\overrightarrow{[\![e]\!]^{\mathcal{V}}}, \vec{x})$$
$$[\![\mu(\overrightarrow{\mathsf{H}[\vec{x}, \vec{\alpha}].c})]\!]^{\mathcal{V}} = \lambda\beta.\beta\ \lambda\gamma.\, \mathbf{case}\, \gamma\, \mathbf{of}\, \overrightarrow{\mathsf{H}(\vec{x}, \vec{\alpha}) \Rightarrow [\![c]\!]^{\mathcal{V}}}$$
$$[\![\tilde{\mu}[\overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}).c}]]\!]^{\mathcal{V}} = \lambda z.\, \mathbf{case}\, z\, \mathbf{of}\, \overrightarrow{\mathsf{K}(\vec{\alpha}, \vec{x}) \Rightarrow [\![c]\!]^{\mathcal{V}}}$$
$$[\![\mathsf{H}_i[\vec{v}, \vec{e}]]\!]^{\mathcal{V}} = \lambda x.[\![v_1]\!]^{\mathcal{V}}\lambda y_1 \ldots . [\![v_n]\!]^{\mathcal{V}}\lambda y_n.x\ \ \mathsf{H}_i(\vec{y}, \overrightarrow{[\![e]\!]^{\mathcal{V}}})$$

Fig. 13. Call-by-value ($\mathcal{V}$) CPS transformation of $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$.

strategy and (co-)data types obtained from their duals equate the dual commands. In addition to recognizing a duality between two hand-crafted strategies and sets of connectives, like for $\mathcal{N}$ and $\mathcal{V}$, this theorem demonstrates a mechanical procedure for generating the semantic dual of *any* strategy and *any* set of (co-)data types, as well as the dual to any theory given as an instance of $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$.

Finally, we need to choose an evaluation strategy for each newly declared (co-)data type. We can do this generically across user-defined (co-)data by deciding on a schema for extending the sets of values and co-values based on (co-)data declarations. For our call-by-value strategy $\mathcal{V}$, we can say that data structures are values if every sub-term is a value, abstract terms are values, and every co-term is a co-value, as shown in Figure 12. This schema agrees with the definition of our call-by-value strategy for all the previously considered (co-)data types, and gives us exactly the same equational theory as we had before. We can provide a schema for our call-by-name evaluation strategy $\mathcal{N}$ in the dual way. In this case, we say that co-structures are co-values if every sub-co-term is a co-value, abstract co-terms are co-values, and every term is a value. Likewise, this schema agrees with the previous definition of our call-by-name strategy.

In addition, we extend the basic call-by-value CPS transformation $[\![\_]\!]^{\mathcal{V}}$ with clauses for the newly declared (co-)data types by encoding (co-)structures and (co-)case abstractions into a CPS λ-calculus extended with user-defined data types, à la ML, as given in Figure 13. The call-by-name CPS transformation is defined as the dual of the call-by-value transformation, $[\![\_]\!]^{\mathcal{N}} = [\![\_^\circ]\!]^{\mathcal{V}}$. It follows that the call-by-value and call-by-name equational theories are sound and complete with respect to the call-by-value and call-by-name CPS transformations, respectively.

Theorem 3. *$\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}} \vdash c = c'$ if and only if $\beta\eta \vdash [\![c]\!]^{\mathcal{S}} = [\![c']\!]^{\mathcal{S}}$, for $\mathcal{S} = \mathcal{V}$ or $\mathcal{N}$.*

The parametric $\mu\tilde{\mu}_{\mathcal{S}}$-calculus extended with user-defined (co-)data types encompasses Wadler's dual sequent calculus [12], where conjunction and disjunction are mapped to the $A \oplus B$ and $A \otimes B$ data types for the call-by-value calculus, and

$$V \in Value_{\mathcal{LV}} ::= x \mid \mathsf{K}_i(\vec{E}, \vec{V}) \mid \mu(\overrightarrow{\mathsf{H}[\vec{x}, \vec{\alpha}].c}) \qquad C_{\mathcal{LV}} \in Context_{\mathcal{LV}} ::= \Box \mid \langle v \| \tilde{\mu} x. C_{\mathcal{LV}} \rangle$$
$$E \in CoValue_{\mathcal{LV}} ::= \alpha \mid \tilde{\mu}[\overrightarrow{\mathsf{K}(\vec{x}, \vec{\alpha}).c}] \mid \mathsf{H}_i[\vec{v}, \vec{E}] \mid \tilde{\mu} x. C_{\mathcal{LV}}[\langle x \| E \rangle]$$

Fig. 14. The $\mathcal{LV}$ strategy for $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$.

$$V \in Value_{\mathcal{LN}} ::= x \mid \mathsf{K}_i(\vec{e}, \vec{V}) \mid \mu(\overrightarrow{\mathsf{H}[\vec{x}, \vec{\alpha}].c}) \mid \mu\alpha. C_{\mathcal{LN}}[\langle V \| \alpha \rangle]$$
$$E \in CoValue_{\mathcal{LN}} ::= \alpha \mid \tilde{\mu}[\overrightarrow{\mathsf{K}(\vec{x}, \vec{\alpha}).c}] \mid \mathsf{H}_i[\vec{V}, \vec{E}] \quad C_{\mathcal{LN}} \in Context_{\mathcal{LN}} ::= \Box \mid \langle \mu\alpha. C_{\mathcal{LN}} \| e \rangle$$

Fig. 15. The $\mathcal{LN}$ strategy for $\mu\tilde{\mu}^{\mathcal{F}}_{\mathcal{S}}$.

to the $A \& B$ and $A \parr B$ co-data types for the call-by-name calculus. Additionally, negation is mapped to the co-data type form of negation for call-by-value, and to the data type form of negation for call-by-name.

Remark 2. So far, we have focused our attention only on two evaluation strategies: the $\mathcal{N}$ strategy for call-by-name and the $\mathcal{V}$ strategy for call-by-value. However, there are other strategies that can be studied by this parametric approach. For instance, we can adapt the "lazy value" strategy [2], $\mathcal{LV}$, to the parametric $\mu\tilde{\mu}_{\mathcal{S}}$-calculus with user-defined (co-)data types as shown in Figure 14. The $\mathcal{LV}$ strategy uses the same notion of value as in $\mathcal{V}$, but restricts co-values to only those co-terms that "need" a value in order to continue. In this way, the $\mathcal{LV}$ behaves in a call-by-name manner by first prioritizing the co-term, and only evaluates terms when demanded. The intuition is that in a context like $\langle v_1 \| \tilde{\mu} y_1. \langle v_2 \| \tilde{\mu} y_2. \Box \rangle \rangle$, v_1 and v_2 are delayed computations whose results are bound to y_1 and y_2 so that they are shared. The co-term $\tilde{\mu} x. \langle v_1 \| \tilde{\mu} y_1. \langle v_2 \| \tilde{\mu} y_2. \langle x \| E \rangle \rangle \rangle$ is strict since $\langle x \| E \rangle$ is the actively running command and it needs the value of x in order to continue, making the whole $\tilde{\mu}$-abstraction a co-value. In the command $\langle \mu\alpha. c_1 \| \tilde{\mu} x. c_2 \rangle$, we begin to evaluate c_2 as if we performed a call-by-name substitution until the value of x is demanded (in a command like $C_{\mathcal{LV}}[\langle x \| E \rangle]$), and then switch to evaluating c_1 by the μ_E rule. The call-by-need $\mathcal{LV}$ strategy demonstrates that there are more than two possible strategies of interest, and that more subtle concerns about evaluation order, such as sharing the results of computations in a non-strict setting, is captured by the parametric $\mu\tilde{\mu}_{\mathcal{S}}$-calculus.

Furthermore, the procedure illustrated by Theorem 2 can be used to mechanically generate a strategy dual to call-by-need, $\mathcal{LN}$, as shown in Figure 15. In this setting, priority is initially given to the producer, but we still share the work needed to reduce a consumer. This strategy may be thought of as call-by-value with a delayed form of control effects, so that a continuation is reduced first before being copied. Delayed control effects introduce new values of the form $\mu\alpha. \langle \mu\beta. \langle V \| \alpha \rangle \| e \rangle$, where we are returning the value V from inside a delayed capture of e. $\mathcal{LN}$ implements the dual form of sharing as call-by-need: it behaves like call-by-value but only captures strict contexts in the sense of call-by-name.

7 Composing multiple strategies

We have now seen how to reason about data and co-data in the sequent calculus according to multiple different evaluation strategies by capturing the essence of the strategy as the parameter to the equational theory. The parameter for the strategy fixes evaluation order once and for all as a global property of the language. However, can we also allow for a program to make use of more than one strategy at a time? Or put another way, can we take several independently consistent strategies and compose them together into a composite strategy for the parametric equational theory, while still maintaining consistency?

The problem calls for a more subtle approach than just taking the union of two or more strategies. For example, if we take the simple union of the call-by-name $\mathcal{N}$ and call-by-value $\mathcal{V}$ strategies, so a (co-)term is a (co-)value if it fits either the $\mathcal{N}$ or $\mathcal{V}$ notions of (co-)value, then the combined strategy considers *every* term and co-term to be a (co-)value. In the command $\langle \mu\alpha.c \| \tilde{\mu}x.c' \rangle$, we could consider the term to be a value by $\mathcal{N}$ criteria and the co-term to be a co-value by $\mathcal{V}$ criteria, leading back to the fundamental inconsistency we were trying to avoid. The issue is that allowing a $\mathcal{N}$ (co-)term interact with a $\mathcal{V}$ (co-)term opens the door for further inconsistencies, even though the two strategies are perfectly consistent in isolation. The solution comes by disallowing (co-)terms from different strategies from interacting directly with one another. If the strategy $\mathcal{S}$ is consistent, then we know that every command is interpreted consistently if we evaluate *both* the term and co-term by $\mathcal{S}$. Our goal is to ensure that every command consistently interprets both term and co-term by the same strategy, and that this consistency is maintained by the rules of the equational theory.

One approach for ensuring that terms only communicate with co-terms following the same strategy is to think about types. We can take all the types which classify programs and put them into different universes, or *kinds*, so that each kind represents one primitive strategy. However, a full static typing discipline is more than necessary for ensuring that programs are consistent. After all, we were able to consistently reason about untyped programs by using a single global strategy, and ideally we would like to keep this property when possible. Therefore, we relax the typing relationship by collapsing all of the types for a particular kind into a single universal type. The notion of having more than one universal type for untyped evaluation is similar to Zeilberger's [14] "bi-typed" system, except that we admit more than two universes, thereby allowing a program to make use of more than two primitive evaluation strategies at a time and in other combinations like call-by-value and call-by-need. To make the strategy for interpreting a (co-)term apparent from its syntax, we explicitly annotate (co-)variables with their kind. An inference system for checking kinds of the core calculus, shown in Figure 16, resembles the type system for $\overline{\lambda}\mu\tilde{\mu}$ [3] except at one level up. The most interesting rule is the cut rule for forming commands that only allows v and e to interact if they both belong to a type of the same kind, ensuring that we interpret v and e according to the same strategy.

In order to allow for user-defined (co-)data types in the presence of multiple primitive strategies, we need to consider kinds when declaring a new (co-)data

$$\frac{}{\Gamma, x :: \mathcal{S} \vdash x^{\mathcal{S}} :: \mathcal{S} | \Delta} \qquad \frac{}{\Gamma | \alpha^{\mathcal{S}} :: \mathcal{S} \vdash \alpha :: \mathcal{S}, \Delta} \qquad \frac{\Gamma \vdash v :: \mathcal{S} | \Delta \quad \Gamma | e :: \mathcal{S} \vdash \Delta}{\langle v \| e \rangle :: (\Gamma \vdash \Delta)}$$

$$\frac{c :: (\Gamma \vdash \alpha :: \mathcal{S}, \Delta)}{\Gamma \vdash \mu\alpha^{\mathcal{S}}.c :: \mathcal{S} | \Delta} \qquad \frac{c :: (\Gamma, x :: \mathcal{S} \vdash \Delta)}{\Gamma | \tilde{\mu}x^{\mathcal{S}}.c :: \mathcal{S} \vdash \Delta}$$

Fig. 16. Type-agnostic kind system for the core calculus.

type. We will illustrate (co-)data declaration with explicit, multiple kinds by example in lieu of presenting the general form. We can declare a strict pair $A \otimes B$, where both components are evaluated eagerly, by annotating the declaration in Figure 8 so that A, B, and $A \otimes B$ belong to the kind $\mathcal{V}$:

$$\mathbf{data}\,(A : \mathcal{V}) \otimes (B : \mathcal{V}) : \mathcal{V}\,\mathbf{where}\ \mathsf{pair} : A : \mathcal{V}, B : \mathcal{V} \vdash A \otimes B : \mathcal{V}|$$

The annotated declaration for $A \otimes B : \mathcal{V}$ introduces the term $\mathsf{pair}(v, v')$ and co-term $\tilde{\mu}[\mathsf{pair}(x^{\mathcal{V}}, y^{\mathcal{V}}).c]$. In addition, the declaration extends the set of $\mathcal{V}$ values with $\mathsf{pair}(V, V')$, where V and V' are $\mathcal{V}$ values, as intended. We are also at liberty to declare a pair using more interesting combinations of strategies, as expressed by kinds. For example, we can introduce a lazy pair $\mathsf{MixedProduct}(A, B)$ of the kind $\mathcal{N}$ where the first component is evaluated strictly:

$$\mathbf{data}\,\mathsf{MixedProduct}(A : \mathcal{V}, B : \mathcal{N}) : \mathcal{N}\,\mathbf{where}$$
$$\mathsf{MixedPair} : A : \mathcal{V}, B : \mathcal{N} \vdash \mathsf{MixedProduct}(A, B) : \mathcal{N}|$$

The declaration of $\mathsf{MixedProduct}(A, B) : \mathcal{N}$ introduces the term $\mathsf{MixedPair}(v, v')$ and co-term $\tilde{\mu}[\mathsf{MixedPair}(x^{\mathcal{V}}, y^{\mathcal{N}}).c]$, both of which are taken to be $\mathcal{N}$ (co-)values. The interesting interplay between the $\mathcal{V}$ and $\mathcal{N}$ strategies in a $\mathsf{MixedProduct}$ is revealed during β reduction:

$$\langle \mathsf{MixedPair}(v, v') \| \tilde{\mu}[\mathsf{MixedPair}(x^{\mathcal{V}}, y^{\mathcal{N}}).c] \rangle =_{\beta, \tilde{\mu}_V} \langle v \| \tilde{\mu}x^{\mathcal{V}}.c \left\{ v'/y^{\mathcal{N}} \right\} \rangle$$

The intended behavior is that after breaking apart the mixed pair, v is evaluated eagerly until it is reduced to a value according to the $\mathcal{V}$ strategy, after which the value is substituted for $x^{\mathcal{V}}$. On the other hand, v' is interpreted according to the $\mathcal{N}$ strategy, so that it is already a value and may be substituted immediately.

Observe that the parametric equational theory instantiated with multiple strategies $\vec{\mathcal{S}}$ and type constructors $\mathcal{F}$, written $\mu\tilde{\mu}^{\mathcal{F}}_{\vec{\mathcal{S}}}$, preserves the well-kindedness of commands and (co-)terms. The axioms in need of the most care are the η axioms, which only apply to variables of the appropriate kind. For instance, the η axiom for $A \otimes B$ of the kind $\mathcal{V}$ only applies to a co-variable $\alpha^{\mathcal{V}}$, whereas the η axiom for $\mathsf{MixedProduct}(A, B)$ of the kind $\mathcal{N}$ only applies to $\alpha^{\mathcal{N}}$.

Theorem 4. *If $\vec{\mathcal{S}}$ are consistent strategies, then $\mu\tilde{\mu}^{\mathcal{F}}_{\vec{\mathcal{S}}}$ is consistent for well-kinded commands and (co-)terms.*

8 Conclusion

The parametric theory provides a direct framework for reasoning about the behavior of programs with both data and co-data in the sequent calculus. We

may understand the meaning of a sequent calculus program using both data structures and message-passing in terms of the intended evaluation strategy. As future work, we would like to develop the theory of the sequent calculus so that it may provide a foundation for objects as a form of co-data, giving us a framework where a notion of object-oriented programming is expressed as the dual paradigm to functional programming. This will involve extending the theory with more advanced features such as inductive and co-inductive forms of self-reference, subtyping, and parametric polymorphism. In addition, we would like to study the suitability of the sequent calculus as an intermediate language in a compiler. Since the sequent calculus provides a framework in which low-level implementation details can be better expressed than in the λ-calculus, we want to study its impact on reasoning about optimizations and program analysis.

Acknowledgments: We would like to thank Pierre-Louis Curien, Hugo Herbelin, and Alexis Saurin for their helpful input and discussion in early versions of this paper, and to acknowledge the support of INRIA and Paris Diderot University while both authors were visiting Paris, where this work was carried out. The authors have also been supported by NSF grant CCF-0917329 and INRIA Équipe Associée SEMACODE.

References

1. A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. POPL '13, 2013.
2. Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In *TLCA*, volume 6690 of *lncs*, 2011.
3. P.-L. Curien and H. Herbelin. The duality of computation. In *International Conference on Functional Programming*, pages 233–243, 2000.
4. P.-L. Curien and G. Munch-Maccagnoni. The duality of computation under focus. *Theoretical Computer Science*, pages 165–181, 2010.
5. A. Filinski. *Declarative Continuations and Categorical Duality.* Master thesis, DIKU, Danmark, Aug. 1989.
6. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
7. H. Herbelin. Duality of computation and sequent calculus: a few more remarks. http://pauillac.inria.fr/ herbelin/publis/full-dual-lk.pdf, 2012.
8. G. Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, pages 409–423. Springer, 2009.
9. S. Ronchi Della Rocca and L. Paolini. *The Parametric λ-Calculus: a Metamodel for Computation.* Springer-Verlag, 2004.
10. P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *MSCS*, 11(2):207–260, 2001.
11. P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of ICFP*, pages 189–201. ACM, 2003.
12. P. Wadler. Call-by-value is dual to call-by-name–reloaded. *Term Rewriting and Applications*, pages 185–203, 2005.
13. N. Zeilberger. On the unity of duality. *Annals of Pure Applied Logic*, 153(1-3):66–96, 2008.
14. N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching.* PhD thesis, Carnegie Mellon University, 2009.