Logical Methods in Computer Science
Volume 16, Issue 3, 2020, pp. 13:1–13:57
https://lmcs.episciences.org/

Submitted Aug. 01, 2019
Published Aug. 28, 2020

COMPILING WITH CLASSICAL CONNECTIVES

PAUL DOWNEN AND ZENA M. ARIOLA

University of Oregon
e-mail address: {pdownen,ariola}@cs.uoregon.edu

Abstract. The study of polarity in computation has revealed that an "ideal" programming language combines both call-by-value and call-by-name evaluation; the two calling conventions are each ideal for half the types in a programming language. But this binary choice leaves out call-by-need which is used in practice to implement lazy-by-default languages like Haskell. We show how the notion of polarity can be extended beyond the value/name dichotomy to include call-by-need by adding a mechanism for sharing which is enough to compile a Haskell-like functional language with user-defined types. The key to capturing sharing in this mixed-evaluation setting is to generalize the usual notion of polarity "shifts:" rather than just two shifts (between positive and negative) we have a family of four dual shifts.

We expand on this idea of logical duality—"and" is dual to "or;" proof is dual to refutation—for the purpose of compiling a variety of types. Based on a general notion of data and codata, we show how classical connectives can be used to encode a wide range of built-in and user-defined types. In contrast with an intuitionistic logic corresponding to pure functional programming, these classical connectives bring more of the pleasant symmetries of classical logic to the computationally-relevant, constructive setting. In particular, an involutive pair of negations bridges the gulf between the wide-spread notions of parametric polymorphism and abstract data types in programming languages. To complete the study of duality in compilation, we also consider the dual to call-by-need evaluation, which shares the computation within the control flow of a program instead of computation within the information flow.

1. Introduction

Finding a universal intermediate language suitable for compiling and optimizing both strict and lazy programs has been a long-sought holy grail for compiler writers. First, there was continuation-passing style (CPS) [Plo75, App92], which hard-codes the evaluation strategy into the program itself. In CPS, all the details of an evaluation strategy can be understood by just looking at the syntax of the program. Second, there were monadic languages [Mog89, PM97], that abstract away from the concrete continuation-passing into a general monadic sequencing operation. Besides moving away from continuations, making them an optional rather than mandatory part of sequencing, they make it easier to incorporate other computational effects by picking the appropriate monad for those effects. Third, there were

Key words and phrases: call-by-need, classical logic, duality, polarity, sequent calculus, type isomorphism.
A previous version of this work appeared in [DA18a].

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.23638/LMCS-16(3:13)2020

adjunctive languages [Lev01, Zei09, MM13], as seen in polarized logic and call-by-push-value λ-calculus, that mix both call-by-name and -value evaluation inside a single program. Like the monadic approach, adjunctive languages make evaluation order explicit within the terms and types of a program, and can easily accommodate effects. However, adjunctive languages also enable more reasoning principles, by keeping the advantages of inductive call-by-value data types, as seen in their denotational semantics. For example, the denotation of a list is just a list of values, not a list of values interspersed with computations that might diverge or cause side effects.

Each of these developments have focused only on call-by-value and -name evaluation, but there are other evaluation strategies out there. For example, to efficiently implement laziness, the Glasgow Haskell Compiler (GHC) uses a core intermediate language which is call-by-need [AMO$^+$95, AF97] instead of call-by-name: the computation of named expressions is shared throughout the lifetime of their result, so that they need not be re-evaluated again. This may be seen as merely an optimization of call-by-name, but it is one that has a profound impact on the other optimizations the compiler can do. For example, full extensionality of functions (*i.e.,* the η law) does not apply in general, due to issues involving divergence and evaluation order. Furthermore, call-by-need is not just a mere optimization but a full-fledged language choice when effects are introduced [AHS11, MM19]: call-by-need and -name are observationally different. This difference may not matter for pure functional programs, but even there, effects *become* important during compilation. For example, it is beneficial to use join points [MDAPJ17], which is a limited form of jump or *goto* statement, to optimize pure functional programs. So it seems like the quest for a common intermediate language for strict and lazy languages that can accommodate effects is still ongoing.

To explore the space of intermediate languages, we use a dual sequent calculus framework which provides a machine-like language suitable for representing functional programs [DMAPJ16]. We begin by reviewing this general framework; specifically:

- (Section 2) We introduce the syntax and reduction theory of $\overline{\lambda}\mu\tilde{\mu}$—a term assignment for Gentzen's sequent calculus LK [Gen35]. We discuss the loss of confluence and extensionality, and how to solve both issues by imposing a global *evaluation strategy*.
- (Section 3) As an alternative to a global evaluation strategy, we discuss a local solution to restore both confluence and extensionality using *polarity*.
- (Section 4) We show how polarity *subsumes* a global evaluation strategy through embeddings of call-by-name and call-by-value calculi into the polarized system, and discuss how a different formulation of the polarity shifting connectives can impact the directness and efficiency of the compiled code.
- (Section 5) To finish the review, we illustrate how sharing can be captured in the sequent calculus through *classical call-by-need* evaluation and its dual.

We then proceed with the goal of integrating these different evaluation strategies together into one cohesive, core calculus. We consider the following questions involving evaluation order and types:

- How can all four evaluation strategies (call-by-value, -name, -need, and -coneed) be combined in the same calculus? What *shifts* are needed to convert between strategies?
- What core connectives are needed to serve as a compile target? Compiling data types is routine, but what is needed to compile *codata types* [Hag87]?
- What is the general notion of user-defined types that can encompass the connectives of both intuitionistic and classical logic?

- How can user-defined data and codata types be compiled into the connectives of the core language, while making sure that the compiled program reflects the full semantics of the source program?
- How can we apply the global compilation technique in a more local manner instead, as an encoding used for optimization? What is the appropriate notion of correctness to ensure that the encoding has exactly the same semantics as the original program, even though the types have changed and may be abstracted and hidden from certain parts of the program? How do we ensure that localized applications of the encoding is robust, even in the presence of computational effects?

This paper answers each of these questions with the following contributions:

- (Section 6) We introduce a family of four shifts, centered around the fundamental call-by-value and call-by-name evaluation strategies, which are capable of correctly compiling the sharing expressed by call-by-(co)need.
- (Section 7) We give a dual core calculus, System $\mathcal{D}$, that has a fixed, finite set of core connectives and can mix all of the above four evaluation strategies within one program.
- (Section 8) We then extend that core calculus with user-defined data and codata types that is fully dual to one another.
- (Section 9) Our notion of user-defined types allows for both parametric polymorphism (in the case of codata types) and a form of abstract data types (*i.e.,* existentially quantified data types), and we show how those defined types can be compiled into the core connectives.
- (Section 10) Finally, we show that this encoding is correct using *type isomorphisms*, which serve as the dual purpose of establishing a correspondence between types and their encodings, as well as illustrating the strong isomorphisms of the core $\mathcal{D}$ connectives that still hold despite the possibility of side effects.

2. Computation in the Classical Sequent Calculus: $\overline{\lambda}\mu\tilde{\mu}$

In the realm of logic, Gentzen's sequent calculus [Gen35] provides a clear window to view the symmetry and duality of classical deduction. And through the Curry-Howard correspondence [Wad15], the sequent calculus can be seen as a programming language for bringing out many symmetries that are otherwise hidden in computation [DA18b]. For example, there are dual calling conventions like call-by-value versus call-by-name [CH00, Wad03] and dual programming constructs like functions versus structures [Zei08, MM09].

One of the ways this computational duality is achieved is by recognizing the two diametrically opposed forces at the heart of every program: the production and the consumption of information. These two forces are formally given a name in the syntax of the sequent calculus: *terms* (denoted by v) produce an output whereas *coterms* (denoted by e) consume an input. For example, a boolean value and a function (written as $\lambda x.v$ in the λ-calculus) are both terms since they produce a result, whereas an if-then-else construct or a call stack (written as $v \cdot e$) are both coterms since they are methods for using an input.

From this viewpoint, computation is modeled as the interaction between a producer and a consumer in the composition $\langle v \| e \rangle$ called a *command* (and denoted by c). Operationally, a command represents the current state of an abstract machine, where the term v acts as program code and the coterm e acts as a continuation. In this way, the sequent calculus serves as a high-level language for abstract machines [ABS09]. However, this language immediately reveals the tension between producers and consumers in computation.

$$
\begin{aligned}
c &::= \langle v \| e \rangle \\
v &::= x \mid \mu\alpha.c \mid \lambda[x \cdot \alpha].c \mid \iota_1 v \mid \iota_2 v \\
e &::= \alpha \mid \tilde{\mu}x.c \mid v \cdot e \mid \mathcal{K}\{\iota_1 x.c_1 \mid \iota_2 y.c_2\}
\end{aligned}
$$

$$
\begin{array}{llll}
(\beta_\mu) & \langle \mu\alpha.c \| e \rangle \to c[e/\alpha] & (\beta_\to) & \left\langle \lambda[x \cdot \alpha].c \middle\| v \cdot e \right\rangle \to c[v/x, e/\alpha] \\
(\beta_{\tilde{\mu}}) & \langle v \| \tilde{\mu}x.c \rangle \to c[v/x] & (\beta_\oplus) & \left\langle \iota_i v \middle\| \mathcal{K}\{\iota_1 x_1.c_1 \mid \iota_2 x_2.c_2\} \right\rangle \to c_i[v/x_i]
\end{array}
$$

Figure 1: The $\overline{\lambda}\mu\tilde{\mu}$-calculus with sums.

2.1. **The fundamental dilemma of computation.** To see why there is a conflict between producers and consumers, consider the small sequent-based language $\overline{\lambda}\mu\tilde{\mu}$ in Figure 1. There are *variables* x representing an unknown term and symmetrically *covariables* representing an unknown coterm. These two identifiers can be bound by $\tilde{\mu}$- and μ-abstractions, respectively. The *input abstraction* $\tilde{\mu}x.c$ binds its input to the variable x before running the command c, and is analogous to the context $\mathbf{let}\, x = \Box \,\mathbf{in}\, c$ where $\Box$ denotes the hole in which the input is placed. Dually, the *output abstraction* $\mu\alpha.c$ binds its partner coterm to α before running c. This can be viewed operationally as "capturing" the current continuation as α analogous to a control operator or the μs of Parigot's $\lambda\mu$-calculus [Par92].

We also have terms and coterms for interacting with specific types of information. The *call stack* $v \cdot e$ is a coterm describing a function call with the argument v such that the returned result will be consumed by e. Call stacks match with a *function abstraction* $\lambda[x \cdot \alpha].c$ which binds the argument and return continuation to x and α, respectively, before running the command c. Values of a sum type can be introduced via one of the two *injections* $\iota_1 v$ or $\iota_2 v$ denoting which side of the sum was chosen. Injections match with a *case abstraction* $\mathcal{K}\{\iota_1 x.c_1 \mid \iota_2 y.c_2\}$ which checks the tag of the injection to determine which of c_1 or c_2 to run.

The operational behavior of this small sequent calculus can be given in terms of substitution (similar to the λ-calculus) as shown in Figure 1. For now consider just the β_μ and $\beta_{\tilde{\mu}}$ reduction rules. These two rules are perfectly symmetric, but already they pose a serious problem: they are completely non-deterministic! To see why, consider the following critical pair where both β_μ and $\beta_{\tilde{\mu}}$ can fire:

$$c_0[\tilde{\mu}x.c_1/\alpha] \leftarrow_{\beta_\mu} \langle \mu\alpha.c_0 \| \tilde{\mu}x.c_1 \rangle \rightarrow_{\beta_{\tilde{\mu}}} c_1[\mu\alpha.c_0/x]$$

There is no reason the two possible reducts must be related. In fact, in the special case where α and x are not used, like in the command $\left\langle \mu\delta.\, \langle x \| \alpha \rangle \middle\| \tilde{\mu}x.\, \langle y \| \beta \rangle \right\rangle$, then we can step to two arbitrarily different results:

$$\langle x \| \alpha \rangle \leftarrow_{\beta_\mu} \left\langle \mu\delta.\, \langle x \| \alpha \rangle \middle\| \tilde{\mu}z.\, \langle y \| \beta \rangle \right\rangle \rightarrow_{\beta_{\tilde{\mu}}} \langle y \| \beta \rangle$$

So this calculus is not just non-deterministic, but also non-confluent (in other words, the starting command above reduces to both $\langle x \| \alpha \rangle$ and $\langle y \| \beta \rangle$, but there is no common c' such that $\langle x \| \alpha \rangle \twoheadrightarrow c' \twoheadleftarrow \langle y \| \beta \rangle$).

2.2. **Global confluence solution: Evaluation strategy.** The loss of confluence is a large departure from similar models of computation like the λ-calculus, and may not be desirable. In Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [CH00], determinism was restored by imposing a discipline onto reduction which prioritized one side over the other:

Sub-syntax and call-by-value reduction rules of $\overline{\lambda}\mu\tilde{\mu}_Q$:

$$c ::= \langle v \| e \rangle$$
$$v ::= V \mid \mu\alpha.c \qquad\qquad V ::= x \mid \lambda[x \cdot \alpha].c \mid \iota_1 V \mid \iota_2 V$$
$$e ::= \alpha \mid \tilde{\mu}x.c \mid V \cdot e \mid \curlywedge\{\iota_1 x.c_1 \mid \iota_2 y.c_2\}$$

$$(\beta^+_\mu) \quad \langle \mu\alpha.c \| e \rangle \to c[e/\alpha] \qquad (\beta^+_\to) \quad \left\langle \lambda[x \cdot \alpha].c \middle\| V \cdot e \right\rangle \to c[V/x, e/\alpha]$$
$$(\beta^+_{\tilde{\mu}}) \quad \langle V \| \tilde{\mu}x.c \rangle \to c[V/x] \qquad (\beta^+_\oplus) \quad \left\langle \iota_i V \middle\| \curlywedge\{\iota_1 x_1.c_1 \mid \iota_2 x_2.c_2\} \right\rangle \to c_i[V/x_i]$$

Sub-syntax and call-by-name reduction rules of $\overline{\lambda}\mu\tilde{\mu}_T$:

$$c ::= \langle v \| e \rangle$$
$$v ::= x \mid \mu\alpha.c \mid \lambda[x \cdot \alpha].c \mid \iota_1 v \mid \iota_2 v$$
$$e ::= E \mid \tilde{\mu}x.c \qquad\qquad E ::= \alpha \mid v \cdot E \mid \curlywedge\{\iota_1 x.c_1 \mid \iota_2 y.c_2\}$$

$$(\beta^-_\mu) \quad \langle \mu\alpha.c \| E \rangle \to c[E/\alpha] \qquad (\beta^-_\to) \quad \left\langle \lambda[x \cdot \alpha].c \middle\| v \cdot E \right\rangle \to c[v/x, E/\alpha]$$
$$(\beta^-_{\tilde{\mu}}) \quad \langle v \| \tilde{\mu}x.c \rangle \to c[v/x] \qquad (\beta^-_\oplus) \quad \left\langle \iota_i v \middle\| \curlywedge\{\iota_1 x_1.c_1 \mid \iota_2 x_2.c_2\} \right\rangle \to c_i[v/x_i]$$

Figure 2: The call-by-value $\overline{\lambda}\mu\tilde{\mu}_Q$ and call-by-name $\overline{\lambda}\mu\tilde{\mu}_T$ sub-calculi.

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

The difference between call-by-value and -name reduction can be seen in λ-calculus terms like $f\ (g\ 1)$: in call-by-value the call $g\ 1$ should be evaluated first before passing the result to f, but in call-by-name the call to f should be evaluated first by passing $g\ 1$ unevaluated as its argument. This is analogous to the sequent calculus command $\left\langle \mu\beta.\langle g \| 1 \cdot \beta \rangle \middle\| \tilde{\mu}x.\langle f \| x \cdot \alpha \rangle \right\rangle$: in call-by-value the μ should be given priority which causes g to be called first, whereas in call-by-name the $\tilde{\mu}$ should be given priority which causes f to be called first. Choosing a priority between μ and $\tilde{\mu}$ effectively restricts the notion of binding and substitution for variables and covariables. Prioritizing β_μ reductions means that μ-abstractions are not substitutable, whereas prioritizing $\beta_{\tilde{\mu}}$ means that $\tilde{\mu}$-abstractions are not substitutable.

In call-by-value, variables stand for a subset of terms called *values* (denoted by V), corresponding to the requirement that arguments are evaluated before being bound to the parameter of a function. The call-by-value evaluation strategy is captured by the restriction of the $\overline{\lambda}\mu\tilde{\mu}$-calculus called $\overline{\lambda}\mu\tilde{\mu}_Q$ [CH00], which introduces a syntax for values as shown in Figure 2. Notice how, in each reduction rule, variables are only ever replaced with values instead of general terms. In a command like $\langle \mu\alpha.c_0 \| \tilde{\mu}x.c_1 \rangle$, only the β^+_μ rule applies because $\mu\alpha.c_0$ is not a value. To support this restriction on reduction, $\overline{\lambda}\mu\tilde{\mu}_Q$ uses a sub-syntax of $\overline{\lambda}\mu\tilde{\mu}$: only values may be pushed onto a call stack ($V \cdot e$) and only values may be injected into a sum ($\iota_i V$). This restriction ensures that the $\beta^+_\to$ and $\beta^+_\oplus$ rules do not get stuck; for example, $\left\langle \lambda[x \cdot \alpha].c \middle\| (\mu\beta.c') \cdot e \right\rangle$ is outside the $\overline{\lambda}\mu\tilde{\mu}_Q$ sub-syntax and is not a $\beta^+_\to$ redex.

Call-by-name evaluation in the sequent calculus follows a dual restriction to call-by-value: covariables stand for a subset of coterms referred to as *covalues* (denoted by E). This corresponds to the intuition that terms are only ever evaluated (via β_μ reduction) when their results are needed. The call-by-name evaluation strategy is captured by the restriction of the $\overline{\lambda}\mu\tilde{\mu}$-calculus called $\overline{\lambda}\mu\tilde{\mu}_T$ [CH00] with an explicit syntax of covalues as shown in Figure 2.

Here, the reduction rules only ever replace covariables with covalues, which analogously forces only the $\beta^-_{\tilde{\mu}}$ rule to apply in the command $\langle \mu\alpha.c_0 \| \tilde{\mu}x.c_1 \rangle$. Likewise, $\overline{\lambda}\mu\tilde{\mu}_T$ uses a different sub-syntax—the tail of all call stacks must be a covalue $(v \cdot E)$—to prevent the $\beta^-_{\to}$ rule from getting stuck; for example, $\big\langle \lambda[x \cdot \alpha].c \big\| v \cdot (\tilde{\mu}x.c') \big\rangle$ is outside the $\overline{\lambda}\mu\tilde{\mu}_T$ sub-syntax and is not a $\beta^-_{\to}$ redex.

2.3. **The impact of evaluation strategy on extensionality.** Similar to the λ-calculus, the above β rules explain how to evaluate commands in the sequent calculus (according to either call-by-value or -name) to get the result of a program. But to reason about programs, the λ-calculus goes beyond just β. The η law $\lambda x.f\ x = f$ captures a notion of extensionality for functions: two functions are the same if they give the same output for every input. This η law can be symmetrically rephrased as follows for both function and sum types in the sequent calculus:

$$(\eta_{\to}) \qquad \lambda[x \cdot \alpha].\langle v \| x \cdot \alpha \rangle \to v \qquad (x \notin FV(v))$$

$$(\eta_{\oplus}) \qquad \mathcal{K}\{\iota_1 x.\langle \iota_1 x \| e \rangle \mid \iota_2 y.\langle \iota_2 y \| e \rangle\} \to e \qquad (\alpha \notin FV(e))$$

These two rules state that matching on an input or output and then rebuilding it exactly as it was is the same thing as doing nothing.

However, it is not always safe to just apply these two η laws as stated above. In fact, when used incorrectly, they can re-introduce the non-determinism that we were seeking to avoid! For example, we have an analogous critical pair in the call-by-value $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus using the $\eta_{\to}$ rule to derive two unrelated results (where x, α, and δ are all unused):

$$\big\langle \lambda[x \cdot \alpha].\langle \mu\delta.c_0 \| x \cdot \alpha \rangle \big\| \tilde{\mu}x.c_1 \big\rangle \to_{\beta^+_{\tilde{\mu}}} c_1$$

$$\big\langle \lambda[x \cdot \alpha].\langle \mu\delta.c_0 \| x \cdot \alpha \rangle \big\| \tilde{\mu}x.c_1 \big\rangle \to_{\eta_{\to}} \langle \mu\delta.c_0 \| \tilde{\mu}x.c_1 \rangle \to_{\beta^+_{\mu}} c_0$$

Therefore, the full $\eta_{\to}$ extensionality rule for functions is not sound in call-by-value. This is not just a problem with functions or call-by-value reduction; a similar critical pair can be constructed in the call-by-name $\overline{\lambda}\mu\tilde{\mu}_T$ using the $\eta_{\oplus}$ rule (where δ and z are unused):

$$\big\langle \mu\delta.c_0 \big\| \mathcal{K}\{\iota_1 x.\langle \iota_1 x \| \tilde{\mu}z.c_1 \rangle \mid \iota_2 y.\langle \iota_2 y \| \tilde{\mu}z.c_1 \rangle\} \big\rangle \to_{\beta^-_{\mu}} c_0$$

$$\big\langle \mu\delta.c_0 \big\| \mathcal{K}\{\iota_1 x.\langle \iota_1 x \| \tilde{\mu}z.c_1 \rangle \mid \iota_2 y.\langle \iota_2 y \| \tilde{\mu}z.c_1 \rangle\} \big\rangle \to_{\eta_{\oplus}} \langle \mu\delta.c_0 \| \tilde{\mu}z.c_1 \rangle \to_{\beta^-_{\tilde{\mu}}} c_1$$

So the soundness of extensional reasoning depends on the evaluation strategy of the language.[1]

3. Polarization: System L

Evaluation strategy can have an impact on the reasoning principles of certain types: the full η law for functions is only sound with call-by-name evaluation, whereas the full η law for sums is only sound with call-by-value evaluation. So does that mean one or the other is doomed to be hamstrung by our inevitable choice? No! Instead of picking just one evaluation strategy for an entire program, we can individually pick and choose which one to apply

[1] Analogous counter-examples appear in the λ-calculus using a non-terminating term (denoted by Ω). The instance $\lambda x.\Omega\ x = \Omega$ of the η law does not hold in the call-by-value λ-calculus. Similarly, the extensionality of sum types in the call-by-value λ-calculus is captured by $\mathbf{case}\, v\, \mathbf{of}\{\iota_1 x \Rightarrow f\ (\iota_1 x) \mid \iota_2 y \Rightarrow f\ (\iota_2 y)\} = f\ v$, but this equality does not hold in the call-by-name λ-calculus when v is Ω and f discards its input.

Sub-syntax with hereditary weak-head normal terms (W) and forcing coterms (F):

$$\begin{array}{ll} c ::= \langle v|s|e\rangle & s ::= + \mid - \\ v ::= W \mid \mu\alpha.c & W ::= x \mid \lambda[x \cdot \alpha].c \mid \iota_1 W \mid \iota_2 W \\ e ::= F \mid \tilde{\mu}x.c & F ::= \alpha \mid W \cdot F \mid \curlywedge\{\iota_1 x.c_1 \mid \iota_2 y.c_2\} \end{array}$$

Definitions of values and covalues in call-by-value (V_+, E_+) and call-by-name (V_-, E_-):

$$V_+ ::= W \qquad E_+ ::= e \qquad V_- ::= v \qquad E_- ::= F$$

Hybrid call-by-value and call-by-name reduction rules:

$$\begin{array}{llll} (\beta^s_\mu) & \langle \mu\alpha.c|s|E_s\rangle \to c[E_s/\alpha] & (\beta_\to) & \big\langle \lambda[x \cdot \alpha].c\big|-\big|W \cdot F\big\rangle \to c[W/x, F/\alpha] \\ (\beta^s_{\tilde{\mu}}) & \langle V_s|s|\tilde{\mu}x.c\rangle \to c[V_s/x] & (\beta_\oplus) & \big\langle \iota_i W\big|+\big|\curlywedge\{\iota_1 x_1.c_1 \mid \iota_2 x_2.c_2\}\big\rangle \to c_i[W/x_i] \end{array}$$

Figure 3: The polarized $\overline{\lambda}\mu\tilde{\mu}$ sub-calculus: System L.

based on the type of information we are dealing with. That way, we can always arrange that the full η laws apply for every type and maximize our ability for extensional reasoning.

3.1. **Local confluence solution: polarity of a command.** Since we will have multiple options for evaluation strategy within the same program, we have to decide which one to use at each reduction. This decision—call-by-value versus call-by-name—might be different at each step but should be statically determined before evaluation, so it must be made apparent in the program itself. And since commands are the only syntactic expressions which can reduce, we only need to know what happens in $\langle v \| e\rangle$: should the interaction between v and e be evaluated according to the call-by-value or call-by-name priority? We can make this decision manifest in the syntax of programs by annotating each command with the sign s denoting its polarity (positive or negative) as $\langle v|s|e\rangle$. This way, s will tell us whether to use the call-by-value priority (for positive $s = +$) or the call-by-name priority (for negative $s = -$). We can formalize this idea of a polarized sequent calculus, called System L, with the grammar in Figure 3. Note that, as far as function and sum types are concerned, this sub-syntax lies in the intersection of both $\overline{\lambda}\mu\tilde{\mu}_T$ and $\overline{\lambda}\mu\tilde{\mu}_Q$. In particular, we have the notion of a *weak-head normal term* W similar to the call-by-value $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus' values, and a *forcing coterm* F similar to the call-by-name $\overline{\lambda}\mu\tilde{\mu}_T$ calculus' covalues. Furthermore, all of the restrictions imposed by $\overline{\lambda}\mu\tilde{\mu}_T$ and $\overline{\lambda}\mu\tilde{\mu}_Q$ are satisfied; injections ι_i can only be applied to weak-head normal terms W, and call stacks can only contain a W and an F. These restrictions can be summarized by the following rule of thumb: constructors and call stacks only apply to terms that immediately return values and coterms that immediately force their output.

Now, consider the semantics for reducing commands of the above polarized System L. Recall that previously we characterized the two different priorities between μ and $\tilde{\mu}$ via the notion of substitutability: which values can replace variables and which covalues can replace covariables. The two different priorities for call-by-value ($+$) and call-by-name ($-$) can now coexist in the same language by the definition of positive and negative values and covalues in Figure 3. With this definition of polarized (co)values, we can now give generic β_μ and $\beta_{\tilde{\mu}}$ reduction rules that perform the correct operation for any given sign s of a command. The other rules for reducing function calls and case analysis are similar as before.

The polarization of a command offers a local solution to the confluence problem discussed in the previous section. For example, the outermost command in $\langle \mu\delta.\, \langle x \| \alpha \rangle \| \tilde{\mu} z.\, \langle y \| \beta \rangle \rangle$ can either be positive, as in $\langle \mu\delta.\, \langle x \| \alpha \rangle |_+ | \tilde{\mu} z.\, \langle y \| \beta \rangle \rangle$ which allows only the call-by-value reduction to $\langle x \| \alpha \rangle$, or negative, as in $\langle \mu\delta.\, \langle x \| \alpha \rangle |_- | \tilde{\mu} z.\, \langle y \| \beta \rangle \rangle$ which allows only the call-by-name reduction to $\langle y \| \beta \rangle$. But it cannot be both. This forces each command to decide which of the two priorities to use, resolving the critical pair in one direction or the other like so:

$$\langle x \| \alpha \rangle \leftarrow_{\beta^+_\mu} \langle \mu\delta.\, \langle x \| \alpha \rangle |_+ | \tilde{\mu} z.\, \langle y \| \beta \rangle \rangle \not\rightarrow_{\beta^+_{\tilde{\mu}}}$$
$$\not\leftarrow_{\beta^-_\mu} \langle \mu\delta.\, \langle x \| \alpha \rangle |_- | \tilde{\mu} z.\, \langle y \| \beta \rangle \rangle \rightarrow_{\beta^-_{\tilde{\mu}}} \langle y \| \beta \rangle$$

However, we must take care that these polarity decisions are made consistently throughout the program. For example, we could have the following critical pair, which relies on a misalignment between the original occurrence of $\mu\delta.c_0$ in a $-$ command and its second reference through the bound variable x in a $+$ command:

$$c_1 \leftarrow_{\beta^-_{\tilde{\mu}}} \langle \mu\delta.c_0 |_- | \tilde{\mu} x.c_1[x/y] \rangle \leftarrow_{\beta^+_\mu} \langle \mu\delta.c_0 |_- | \tilde{\mu} x.\, \langle x |_+ | \tilde{\mu} y.c_1 \rangle \rangle \rightarrow_{\beta^-_{\tilde{\mu}}} \langle \mu\delta.c_0 |_+ | \tilde{\mu} y.c_1 \rangle \rightarrow_{\beta^+_\mu} c_0$$

where again, δ, x, and y do not occur free in c_0 and c_1. So to restore confluence, we need to ensure that misaligned signs are ruled out. But before presenting the full solution to this issue, we examine how extensionality interacts with polarity.

3.2. **The impact of polarity on extensionality.** The goal of combining both call-by-value and call-by-name reduction, as above, is so that we could safely validate the full η axioms for both the function and sum types. We can now explicitly enforce the requirement that each axiom only applies to terms or coterms following a specific evaluation strategy with $+$ and $-$ sign annotations like so:

$$(\eta^-_\to) \qquad \lambda[x \cdot \alpha].\, \langle v |_- | x \cdot \alpha \rangle \to v$$
$$(\eta^+_\oplus) \qquad \text{Y}\{\iota_1 x.\langle \iota_1 x |_+ | e \rangle \mid \iota_2 y.\langle \iota_2 y |_+ | e \rangle\} \to e$$

Notice how, so long as the signs in the command align—meaning that each term and coterm consistently appears in only positive or only negative commands—we prevent the possibility of non-confluence. For example, the following command is confluent due to this alignment, since only the call-by-name $\beta^-_{\tilde{\mu}}$ rule applies both before and after η-reduction:

$$c_1 \leftarrow_{\beta^-_{\tilde{\mu}}} \langle \lambda[x \cdot \alpha].\, \langle \mu\delta.c_0 |_- | x \cdot \alpha \rangle |_- | \tilde{\mu} x.c_1 \rangle \rightarrow_{\eta^-_\to} \langle \mu\delta.c_0 |_- | \tilde{\mu} x.c_1 \rangle \rightarrow_{\beta^-_{\tilde{\mu}}} c_1$$

where again we assume that x, α, and δ are not used. In contrast, the counter-example to confluence with $\eta^-_\to$ has mismatching sign annotations like so:

$$c_1 \leftarrow_{\beta^+_{\tilde{\mu}}} \langle \lambda[x \cdot \alpha].\, \langle \mu\delta.c_0 |_- | x \cdot \alpha \rangle |_+ | \tilde{\mu} x.c_1 \rangle \rightarrow_{\eta^-_\to} \langle \mu\delta.c_0 |_+ | \tilde{\mu} x.c_1 \rangle \rightarrow_{\beta^+_\mu} c_0$$

The misalignment between signs is due to the fact that a function abstraction is a negative term, but it is used in a positive command.

Determining the polarity of a type:

$$A, B, C ::= X \mid \overline{X} \mid A \to B \mid A \oplus B$$

$$\frac{}{X : +} \qquad \frac{}{\overline{X} : -} \qquad \frac{A : + \quad B : +}{A \oplus B : +} \qquad \frac{A : + \quad B : -}{A \to B : -}$$

Inference rules that are generic for any type:

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

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

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

$$\frac{\Gamma \vdash W : A \,;\, \Delta}{\Gamma \vdash W : A \mid \Delta}\ FR \qquad \frac{\Gamma \,;\, F : A \vdash \Delta}{\Gamma \mid F : A \vdash \Delta}\ FL$$

Inference rules for specific types:

$$\frac{c : (\Gamma, x : A \vdash \alpha : B, \Delta)}{\Gamma \vdash \lambda[x \cdot \alpha].c : A \to B \,;\, \Delta} \to R \qquad \frac{\Gamma \vdash W : A \,;\, \Delta \quad \Gamma \,;\, F : B \vdash \Delta}{\Gamma \,;\, W \cdot F : A \to B \vdash \Delta} \to L$$

$$\frac{\Gamma \vdash W : A \,;\, \Delta}{\Gamma \vdash \iota_1 W : A \oplus B \,;\, \Delta} \oplus R_1 \qquad \frac{\Gamma \vdash W : B \,;\, \Delta}{\Gamma \vdash \iota_2 W : A \oplus B \,;\, \Delta} \oplus R_2$$

$$\frac{c_1 : (\Gamma, x : A \vdash \Delta) \quad c_2 : (\Gamma, y : B \vdash \Delta)}{\Gamma \,;\, \curlywedge\{\iota_1 x.c_1 \mid \iota_2 y.c_2\} : A \oplus B \vdash \Delta} \oplus L$$

Figure 4: Type system for System L.

3.3. **Polarity in the type system.** One way to ensure that the signs in a program are consistently aligned is to link polarity with types. In other words, we can distinguish the two different kinds of types according to their polarity: positive types (of kind $+$) following call-by-value evaluation and negative types (of kind $-$) following call-by-name evaluation. For example, consider the basic grammar of types which includes functions ($A \to B$) and sum ($A \oplus B$) types given in Figure 4. For the base cases, we also have atomic type variables of positive (X) and negative ($\overline{X}$) kinds. The distinction between positive and negative types is given by the inference rules for checking $A : +$ versus $A : -$. Since the full η law for sum types requires call-by-value evaluation, $A \oplus B$ is positive, and dually since the full η law for function types requires call-by-name evaluation, $A \to B$ is negative. Also note that the polarity of a type is hereditary. The types A and B inside $A \oplus B$ must also be positive. Interestingly, there is some added subtlety to functions, the return type B of $A \to B$ must also be negative, but the input type A is instead positive. The reason that the polarity of function inputs are reversed is due to the reversal of information flow caused by a function: outputs have the same polarity as the connective, whereas inputs have the opposite polarity.

The type system for System L which ensures confluence is given in Figure 4. Since there are several different syntactic categories in the language, we have the following different typing judgments:

- $c : (\Gamma \vdash \Delta)$ says that the command c is well-typed, assuming the typing assignments for variables in the environment Γ and covariables in the environment Δ.[2]
- $\Gamma \vdash v : A \mid \Delta$ says that the term v eventually produces an output of type A.
- $\Gamma \vdash W : A \mathbin{;} \Delta$ says that W is already a value of type A.
- $\Gamma \mid e : A \vdash \Delta$ says that the coterm e eventually consumes an input of type A.
- $\Gamma \mathbin{;} F : A \vdash \Delta$ says that F is already a covalue of type A.

The *Cut* rule forms a command between a term and coterm of the same type A, while also ensuring that the sign s in the command matches the kind of A. The *VR* and *VL* rules refer to a (co)variable in the environment, and the *AR* and *AL* rules bind a (co)variable with a μ- or $\tilde{\mu}$-abstraction, respectively. And the *FR* and *FL* rules correspond to the inclusion of W in terms and F in coterms. As a result, all well-typed commands are confluent, and all well-typed expressions (commands, terms, and coterms) are confluent. For example, the troublesome command above with misaligned signs is not typable in this system, as shown by the following derivation that *fails* on one of the highlighted premises $A : -$ or $A : +$, because there is no type A that is both positive and negative:

$$\dfrac{\dfrac{c_0 : (\Gamma \vdash \delta : A, \Delta)}{\Gamma \vdash \mu\delta.c_0 : A \mid \Delta}\, AR \qquad A : -}{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash x : A \mathbin{;} \Delta}\, VR}{\Gamma, x : A \vdash x : A \mid \Delta}\, FR \qquad A : +}{\langle x|_+|\tilde{\mu}y.c_1\rangle : (\Gamma, x : A \vdash \Delta)}\, AL \qquad \dfrac{c_1 : (\Gamma, x : A, y : A \vdash \Delta)}{\Gamma, x : A \mid \tilde{\mu}y.c_1 : A \vdash \Delta}\, AL}{\Gamma \mid \tilde{\mu}x.\langle x|_+|\tilde{\mu}y.c_1\rangle : A \vdash \Delta}\, Cut}{\langle \mu\delta.c_0 |_-| \tilde{\mu}x.\langle x|_+|\tilde{\mu}y.c_1\rangle \rangle : (\Gamma \vdash \Delta)}\, Cut$$

Intermezzo 3.1. We used different judgments for typing general terms versus weak-head normal terms ($\Gamma \vdash v : A \mid \Delta$ versus $\Gamma \vdash W : A \mathbin{;} \Delta$) and general coterms versus forcing coterms. One motivation for doing so is to capture the notion of sub-syntax in the polarized System L inside the type system itself: the fact that a call stack must consist of a W and an F can be read off from the judgments in the $\to L$ inference rule, even if we erased the terms and coterms. It turns out that the impact of these syntactic restrictions on typing corresponds to the notion of *focusing* in logic [And92], which was originally developed to aid proof search. The *focused* judgments $\Gamma \vdash W : A \mathbin{;} \Delta$ and $\Gamma \mathbin{;} F : A \vdash \Delta$ are more restrictive than their counterpart $\Gamma \vdash v : A \mid \Delta$ and $\Gamma \mid e : A \vdash \Delta$. For example, there is no way to derive $\Gamma \vdash \mu\alpha.c : A \mathbin{;} \Delta$, even if $\Gamma \vdash \mu\alpha.c : A \mid \Delta$ is derivable. This limits the number of derivations that can be formed, in the same way that the various sub-syntaxes limit the number of programs that can be written.

3.4. **Restoring expressiveness with polarity shifts.** Unfortunately, it turns out that just distinguishing the two kinds of types, as in the type system above, severely weakens the system. The identity function $A \to A$ is no longer well-formed! If A is positive, then it cannot be the return type of a function, and if A is negative, it cannot be the input type. This limitation is clearly untenable, and so we need *polarity shifts* which explicitly coerce between the two kinds of types: the down-shift $\downarrow$ converts a negative type to a positive type, and the up-shift $\uparrow$ converts a positive type in a negative type. For example, we could write the identity function as either $\downarrow A \to A$, when A is negative, or $A \to \uparrow A$, when A is positive. With these polarity shifts, the expressiveness of the system is restored.

[2] In each of these judgments, the environment $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a set of variable-type pairs and $\Delta = \alpha_1 : A_1, \ldots, \alpha_n : A_n$ is a set of covariable-type pairs, such that all (co)variables are distinct in both.

Extending the syntax of types and programs with shifts:

$$A, B, C ::= \ldots \mid \downarrow A \mid \uparrow B \qquad W ::= \ldots \mid \downarrow v \mid \lambda\uparrow\alpha.c \qquad F ::= \ldots \mid \lambda\downarrow x.c \mid \uparrow e$$

Reduction rules for shifts:

$$(\beta_\downarrow) \qquad \langle \downarrow v \mid_+ \mid \lambda\downarrow x.c \rangle \to c[v/x] \qquad (\beta_\uparrow) \qquad \langle \lambda\uparrow\alpha.c \mid_- \mid \uparrow e \rangle \to c[e/\alpha]$$

Typing rules for shifts:

$$\frac{A : -}{\downarrow A : +} \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \downarrow v : \downarrow A \mid \Delta}\ \downarrow R \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \lambda\downarrow x.c : \downarrow A \vdash \Delta}\ \downarrow L$$

$$\frac{A : +}{\uparrow A : -} \qquad \frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \lambda\uparrow\alpha.c : \uparrow A \mid \Delta}\ \uparrow R \qquad \frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \uparrow e : \uparrow A \vdash \Delta}\ \uparrow L$$

Figure 5: Polarity shifts in System L.

System L can be extended with the shifts given in Figure 5. The terms and coterms for the polarity shifts are similar to those for function and sum types. The up-shift $\uparrow A$ is analogous to a nullary function $1 \to A$ (where 1 is a unit type) with the exception that A is positive, rather than negative. As such, it contains a λ-abstraction of the form $\lambda\uparrow\alpha.c$ (with no argument parameter x) and a call stack of the form $\uparrow e$ (with no restriction on e because all coterms force their input in call-by-value). The down-shift $\downarrow A$ is analogous to a degenerate sum type $0 \oplus A$ (where 0 is an empty type) with the exception that A is negative, rather than positive. It contains a single injection $\downarrow v$ (with no restriction on v because all terms are already values in call-by-name) and a pattern-matching abstraction of the form $\lambda\downarrow x.c$ for unwrapping the injection.

4. Compiling Call-by-Name and Call-by-Value to System L

With the polarity shifts between positive and negative types, we can express every program that could have been written in the unpolarized languages. But now since *both* call-by-value and -name evaluation is denoted by different types, the types themselves signify the calling convention. Both compilations of $\overline{\lambda}\mu\tilde{\mu}_Q$ and $\overline{\lambda}\mu\tilde{\mu}_T$ into System L are shown in Figure 6. In the call-by-value compilation of $\overline{\lambda}\mu\tilde{\mu}_Q$, every type is translated to a positive type, in other words $A^+ : +$. This leaves the sum connective $\oplus$ since it is already purely positive, but the function type requires two shifts: one $\uparrow$ to convert the positive return type of the function to a negative type (satisfying the preconditions of the polarized function arrow) and another $\downarrow$ to ensure the overall function type itself is positive instead of negative. In contrast, the call-by-name compilation of $\overline{\lambda}\mu\tilde{\mu}_T$ translates every type to a negative type, in other words $A^- : -$. As a result, the call-by-name compilation of a function requires only one shift to convert the input type of the function to be positive, but the sum connective now requires heavy shifting: two $\downarrow$ shifts to convert both sides of the sum to be positive (satisfying the preconditions of the polarized sum) and a $\uparrow$ shift to make the overall sum type itself negative.

At a basic level, these two compilations make sense from the perspective of typability (corresponding to provability in logic)—by inspection, all of the types line up with their newly-assigned polarities. But programs are meant to be run, so we care about more than just typability. At a deeper level, the compilations are *sound* with respect to equality of

Call-by-value polarizing compilation of $\overline{\lambda}\mu\tilde{\mu}_Q$ into System L.

$$X^+ \triangleq X \qquad (A \to B)^+ \triangleq \downarrow(A^+ \to (\uparrow B^+)) \qquad (A \oplus B)^+ \triangleq A^+ \oplus B^+$$

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

$$\begin{aligned} x^+ &\triangleq x & \alpha^+ &\triangleq \alpha \\ (\mu\alpha.c)^+ &\triangleq \mu\alpha.(c^+) & (\tilde{\mu}x.c)^+ &\triangleq \tilde{\mu}x.(c^+) \\ (\lambda[x \cdot \alpha].c)^+ &\triangleq \downarrow\big(\lambda[x \cdot \beta].\langle \lambda\uparrow\alpha.c^+ | - | \beta \rangle\big) & (V \cdot e)^+ &\triangleq \mathcal{K}\downarrow x.\langle x | - | V^+ \cdot [\uparrow e^-] \rangle \\ (\iota_i V)^+ &\triangleq \iota_i(V^+) & \mathcal{K}\{\iota_i x_i.c_i\}^+ &\triangleq \mathcal{K}\{\iota_i x_i.c_i^+\} \end{aligned}$$

Call-by-name polarizing compilation of $\overline{\lambda}\mu\tilde{\mu}_T$ into System L.

$$X^- \triangleq \overline{X} \qquad (A \to B)^- \triangleq (\downarrow A^-) \to B^- \qquad (A \oplus B)^- \triangleq \uparrow((\downarrow A^-) \oplus (\downarrow B^-))$$

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

$$\begin{aligned} x^- &\triangleq x & \alpha^- &\triangleq \alpha \\ (\mu\alpha.c)^- &\triangleq \mu\alpha.(c^-) & (\tilde{\mu}x.c)^- &\triangleq \tilde{\mu}x.(c^-) \\ (\lambda[x \cdot \alpha].c)^- &\triangleq \lambda[y \cdot \alpha].\langle y | + | \mathcal{K}\downarrow x.c^- \rangle & (v \cdot E)^- &\triangleq (\downarrow v^-) \cdot E^- \\ (\iota_i v)^- &\triangleq \lambda\uparrow\beta.\langle \iota_i(\downarrow v^-) | + | \beta \rangle & \mathcal{K}\{\iota_i x_i.c_i\}^- &\triangleq \uparrow\big[\mathcal{K}\{\iota_i y_i.\langle y_i | + | \mathcal{K}\downarrow x_i.c_i^- \rangle\}\big] \end{aligned}$$

Figure 6: Compiling $\overline{\lambda}\mu\tilde{\mu}_Q$ and $\overline{\lambda}\mu\tilde{\mu}_T$ into System L.

terms: if two terms are equal, then their compiled forms are also equal. We have not yet formally defined equality, so we will return to this question later in Section 10.

4.1. **More direct compilation with four shifts.** However, even though these compilations are correct, some of the cases are much more direct than others. In particular, observe how the negative compilation of functions can be understood as just expanding a nested copattern match like so:

$$(\lambda[x \cdot \alpha].c)^- = \lambda[\downarrow x \cdot \alpha].c^-$$

which matches the macro-expanded call stack $(v \cdot E)^- = (\downarrow v^-) \cdot E^-$. However, the positive compilation of functions is not so nice; it wraps a λ-abstraction in a $\downarrow$ shift, which means that a $\overline{\lambda}\mu\tilde{\mu}_Q$ call stack of the form $V \cdot e$ is hidden inside a pattern-matching abstraction in order to extract the contents of the shifted function. Similarly, the positive compilation of sum types just distributes over constructors and pattern matching, since sum types are already purely positive, but the negative compilation of a sum injection $(\iota_i v)^-$ is hidden inside a λ-abstraction for the same reason.

For now, this mismatch between construction and pattern matching is tolerable, but when considering other compilations (Section 5) it will become unacceptable. The issue in both cases is with the outermost shift: the constructor or destructor is applied to the wrong side (term versus coterm) which forces us to wrap a λ-abstraction instead of a call stack or sum injection. Fortunately, there is an easy way to remedy the situation: use a different formulation of the two shifts.

Notice how—as a side effect of the fact that $\downarrow v$ can contain any term v and $\uparrow e$ can contain any coterm e—the typing rules $\downarrow R$ and $\uparrow L$ *lose* focus (see Intermezzo 3.1) since

Extending the syntax of types and programs with shifts:

$$A, B, C ::= \ldots \mid \Uparrow A \mid \Downarrow B \qquad W ::= \ldots \mid \Uparrow W \mid \lambda\Downarrow\alpha.c \qquad F ::= \ldots \mid \mathcal{K}\Uparrow x.c \mid \Downarrow F$$

Reduction rules for shifts:

$$(\beta_{\Downarrow}) \qquad \langle \lambda\Downarrow\alpha.c \mid_{+} \mid \Downarrow F\rangle \to c[F/\alpha] \qquad (\beta_{\Uparrow}) \qquad \langle \Uparrow W \mid_{-} \mid \mathcal{K}\Uparrow x.c\rangle \to c[W/x]$$

Typing rules for shifts:

$$\frac{A : {}_{-}}{\Downarrow A : {}_{+}} \qquad \frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \lambda\Downarrow\alpha.c : \Downarrow A \,;\, \Delta}\ \Downarrow R \qquad \frac{\Gamma \,;\, F : A \vdash \Delta}{\Gamma \,;\, \Downarrow F : \Downarrow A \vdash \Delta}\ \Downarrow L$$

$$\frac{A : {}_{+}}{\Uparrow A : {}_{-}} \qquad \frac{\Gamma \vdash W : A \,;\, \Delta}{\Gamma \vdash \Uparrow W : \Uparrow A \,;\, \Delta}\ \Uparrow R \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \,;\, \mathcal{K}\Uparrow x.c : \Uparrow A \vdash \Delta}\ \Uparrow L$$

Figure 7: Alternative polarity shifts in System L.

their premise is the unfocused judgment $\Gamma \vdash v : A \mid \Delta$ and $\Gamma \mid e : A \vdash \Delta$, respectively. These are the first rules with this property: all the other typing rules with focused conclusions have focused premises when possible. The only exceptions to this generalization are rules like $\to R$ and $\oplus L$, because commands do not have any notion of focus like terms and coterms do. It turns out, there is an alternative way to formulate the shifts so that focus is *gained* rather than *lost*. In other words, we could have defined polarity shifts that follow the rule of thumb where constructors and destructors only apply to W terms and F coterms.

These alternative shifts can be added to the calculus, as shown in Figure 7, where the up-shift $\Uparrow$ has the same role of converting positive to negative, and the down-shift $\Downarrow$ still converts from negative to positive. The constructor $\Uparrow$ only applies to a W, and $\Downarrow$ only applies to a F. Instead, the change in polarity is captured by the fact that $\lambda\Downarrow\alpha.c$ is a positive value distinct from the non-value $\mu\alpha.c$, and $\mathcal{K}\Uparrow x.c$ is a negative covalue which forces its input unlike the non-strict $\tilde{\mu}x.c$. Note that it is important for the conclusions of the $\Downarrow R$ and $\Uparrow L$ rule be in focus (*i.e.,* use the stoup separator ; rather than |). Otherwise, appearances of the thunk-like abstraction $\lambda\Downarrow\alpha.c$ in an injection $(\iota_i(\lambda\Downarrow\alpha.c))$ or call stack $((\lambda\Downarrow\alpha.c) \cdot F)$ would not type-check. But the premises of these rules are commands, like in the $\to R$ and $\oplus L$ rules, which cannot be in focus. From this standpoint, the thick shifts ($\Downarrow$ and $\Uparrow$) have different focusing properties from the thin shifts ($\downarrow$ and $\uparrow$).

The compilation translations in Figure 6 can now be cleaned up with the following alternative compilation using both styles of shifts, whose impact on terms and coterms is completely captured as a macro-expansion of patterns and copatterns like so:

$$(A \to B)^{+} = \Downarrow(A^{+} \to (\uparrow B^{+})) \qquad (V \cdot e)^{+} = \Downarrow[W^{+} \cdot [\uparrow e^{+}]] \qquad (\lambda[x \cdot \alpha].c)^{+} = \lambda\Downarrow[x \cdot [\uparrow\alpha]].c^{+}$$

$$(A \oplus B)^{-} = \Uparrow((\downarrow A^{-}) \oplus (\downarrow B^{-})) \qquad (\iota_i v)^{-} = \Uparrow(\iota_i(\downarrow v^{-})) \qquad \mathcal{K}\{\iota_i x_i.c_i\}^{-} = \mathcal{K}\{\Uparrow(\iota_1(\downarrow x_i)).c_i^{-}\}$$

Note that the direction of the shifts are the same as before, the only thing that changes is that the outer arrows are thick ($\Downarrow$ or $\Uparrow$), whereas the inner arrows are thin ($\downarrow$ or $\uparrow$). While this is just an aesthetic change for now, the impact of the two different styles of shifts in compilation becomes crucial once we go beyond just call-by-value and call-by-name evaluation.

5. Classical Call-by-Need and its Dual

Polarity neatly integrates the best of both call-by-value and call-by-name languages. However, there is more to computation than just those two evaluation strategies. Call-by-need evaluation is a different way to integrate aspects of both call-by-name and -value: like call-by-name results are only computed when they are needed, but like call-by-value they are never evaluated more than once. For example, let $I = \lambda x.x$ be the identity function, so that f will never be evaluated in the expression $\mathbf{let}\, f = I\ I\, \mathbf{in}\, 5$ because it is not needed to return the result 5. In contrast, f will be evaluated exactly once in $\mathbf{let}\, f = I\ I\, \mathbf{in}\, I\ f\ f\ 5$ like so:

$$\begin{aligned}\mathbf{let}\, f = I\ I\, \mathbf{in}\, \underline{I\ f}\ f\ 5 &\to \mathbf{let}\, f = \underline{I\ I}\, \mathbf{in}\, f\ f\ 5\\ &\to \underline{\mathbf{let}\, f = I\, \mathbf{in}\, f\ f\ 5}\\ &\to \underline{I\ I}\ 5 \to \underline{I\ 5} \to 5\end{aligned}$$

where the next reduction to take place at each step is underlined for clarity. Notice how, in order to simultaneously delay and share the computation of f, reduction occurs *underneath* **let** bindings. First, the inner redex $I\ f$ is performed, after which f is needed to continue evaluating the expression $f\ f\ 5$ which forces the evaluation of f. The ability to reduce inside of **let**s is the key to the call-by-need λ-calculus [AMO+95, AF97].

5.1. **Sharing work in the sequent calculus.** If delayed **let** binding is the key to the call-by-need λ-calculus, what is the key to a call-by-need sequent calculus? Since $\tilde{\mu}$-abstractions are the analogue to **let**s, a call-by-need priority between β_μ and $\beta_{\tilde{\mu}}$ reductions relies on delayed $\tilde{\mu}$-bindings [AHS11]. Rather than forcing one substitution or the other in the command $\langle \mu\beta.c' \| \tilde{\mu}x.c \rangle$ like call-by-value and call-by-name do, call-by-need reduces the command c underneath the $\tilde{\mu}$-binding of x. In general, this reduction will proceed like so:

$$\langle \mu\beta.c' \| \tilde{\mu}x.\underline{c} \rangle \twoheadrightarrow \left\langle \mu\beta.c' \middle\| \tilde{\mu}x.\, \langle v_1 \| \tilde{\mu}y_1. \ldots \langle v_n \| \tilde{\mu}y_n.\, \langle x \| E \rangle \rangle \rangle \right\rangle$$

where E is some covalue which forces the evaluation of the bound variable x. Note how there may be many more delayed $\tilde{\mu}$-bindings allocated during this evaluation step, which might be referenced inside E. At this point, no more progress can be made under the $\tilde{\mu}$-binding of x, so that the whole $\tilde{\mu}$-abstraction is done. We must now refocus our attention to the term bound to x, which is modeled by finally now performing a β_μ on the command:

$$\left\langle \mu\beta.c' \middle\| \tilde{\mu}x.\, \langle v_1 \| \tilde{\mu}y_1. \ldots \langle v_n \| \tilde{\mu}y_n.\, \langle x \| E \rangle \rangle \rangle \right\rangle \to_{\beta_\mu} c'[\tilde{\mu}x.\, \langle v_1 \| \tilde{\mu}y_1. \ldots \langle v_n \| \tilde{\mu}y_n.\, \langle x \| \alpha \rangle \rangle \rangle / \beta]$$

This step allows for the term $\mu\beta.c'$ to be evaluated by running the command c' with a substitution for β. As c' runs, a value may be "returned" to its consumer β like so

$$\left\langle V \middle\| \tilde{\mu}x.\, \langle v_1 \| \tilde{\mu}y_1. \ldots \langle v_n \| \tilde{\mu}y_n.\, \langle x \| \alpha \rangle \rangle \rangle \right\rangle \to_{\beta_{\tilde{\mu}}} \langle v_1[V/x] \| \tilde{\mu}y_1. \ldots \langle v_n[V/x] \| \tilde{\mu}y_n.\, \langle V \| \alpha \rangle \rangle \rangle$$

Since β may be used many times, several values may be "returned" to the consumer of $\mu\beta.c'$ along β, but c' is run at most once: this is the essence of classical call-by-need evaluation.

With this intuition, we can simulate the same example above from the call-by-need λ-calculus using the more machine-like sequent calculus, where the identity function is written as $\lambda[x \cdot \alpha].\, \langle x \| \alpha \rangle$. This example illustrates the back-and-forth reductions caused by

evaluating underneath $\tilde{\mu}$-bindings.

$$\begin{aligned}\left\langle \mu\beta.\,\langle I \| I \cdot \beta\rangle \middle\| \tilde{\mu} f.\underline{\langle I \| f \cdot f \cdot 5 \cdot \alpha\rangle} \right\rangle &\to_{\beta_{\to}} \underline{\left\langle \mu\beta.\,\langle I \| I \cdot \beta\rangle \middle\| \tilde{\mu} f.\,\langle f \| f \cdot 5 \cdot \alpha\rangle \right\rangle} \\ &\to_{\beta_{\mu}} \underline{\left\langle I \middle\| I \cdot \tilde{\mu} f.\,\langle f \| f \cdot 5 \cdot \alpha\rangle \right\rangle} \\ &\to_{\beta_{\to}} \underline{\left\langle I \middle\| \tilde{\mu} f.\,\langle f \| f \cdot 5 \cdot \alpha\rangle \right\rangle} \\ &\to_{\beta_{\tilde{\mu}}} \underline{\langle I \| I \cdot 5 \cdot \alpha\rangle} \to_{\beta_{\to}} \underline{\langle I \| 5 \cdot \alpha\rangle} \to_{\beta_{\to}} \underline{\langle 5 \| \alpha\rangle}\end{aligned}$$

Notice how the expression $\mu\beta.\,\langle I \| I \cdot \beta\rangle$ is delayed in the first step but never copied, even though the bound variable f is used multiple times. Also notice how every reduction step, besides the first one, occurs at the top of the command.

5.2. **Classical call-by-need.** Since the sequent calculus corresponds to Gentzen's system LK of classical logic, defining call-by-need evaluation for the sequent calculus gives a notion of "classical call-by-need," meaning call-by-need evaluation with first-class control effects [Gri90]. Even though call-by-need is a more intricate evaluation strategy than call-by-value and -name, it can be formulated using the same technique: managing the priority between producers and consumers through a notion of substitutability. The only question is, which terms are substitutable values, and dually which coterms are covalues?

Substitutability means that values can be copied and deleted, and on this front both call-by-value and call-by-need agree. General μ-abstractions, which represents arbitrary computation, cannot be copied in either evaluation strategies. Instead, λ-abstractions—which are first-class values—can be copied, and so too can injections of the form $\iota_i V$ where V is another value. The fact that $\iota_i v$ is not a value in call-by-need corresponds to the fact that the arguments to constructors are shared, not computed more than once. In contrast, the notion of covalues in call-by-need corresponds to coterms which "force" their input. This includes all the call-by-name forms of covalues, but this alone is not enough.

In the previous example of sharing in the sequent calculus, notice how we were forced to substitute $\tilde{\mu} f.\,\langle f \| f \cdot 5 \cdot \alpha\rangle$ in the step

$$\left\langle \mu\beta.\,\langle I \| I \cdot \beta\rangle \middle\| \tilde{\mu} f.\,\langle f \| f \cdot 5 \cdot \alpha\rangle \right\rangle \to_{\beta_{\mu}} \left\langle I \middle\| I \cdot \tilde{\mu} f.\,\langle f \| f \cdot 5 \cdot \alpha\rangle \right\rangle$$

If $\tilde{\mu}$-abstractions could never be substituted like in call-by-name, then this step would not be allowed and the computation would get stuck. But we can see that this specific instance is okay, because the input (named f by the $\tilde{\mu}$-abstraction) is immediately needed in its body. In general, $\tilde{\mu}$-abstractions of the form $\tilde{\mu} x.\,\langle v_1 \| \tilde{\mu} y_1. \ldots \langle v_n \| \tilde{\mu} y_n.\,\langle x \| E\rangle\rangle\rangle$ (where x is different than $y_1 \ldots y_n$) forces its input, because the bound x is immediately consumed by another forcing covalue E.

To make sure that computation does not get stuck, $\tilde{\mu}$-abstractions of this form must be considered substitutable covalues, which gives us the sub-syntax for call-by-need $\overline{\lambda}\mu\tilde{\mu}$ shown in Figure 8. Intuitively, the set of contexts H represent a heap of variables bound to thunks (potentially unevaluated terms). Note that there is a side condition for covalues of the form $\tilde{\mu} x.H[\langle x \| E\rangle]$: H must not contain a binding for x which is in scope over its hole $\Box$. With this sub-syntax, we can use the reduction rules given in Figure 8 for performing call-by-need evaluation. Notice how the general form of the reduction rules are the same as in the other call-by-name, call-by-value, and polarized calculi; the only real difference is the definition of V and E.

Sub-syntax for call-by-need evaluation:

$$\begin{array}{ll} c ::= \langle v \| e \rangle & H ::= \Box \mid \langle v \| \tilde{\mu} x.H \rangle \\ v ::= V \mid \mu\alpha.c & V ::= x \mid \lambda[x \cdot \alpha].c \mid \iota_1 V \mid \iota_2 V \\ e ::= E \mid \tilde{\mu} x.c & E ::= \alpha \mid V \cdot E \mid \mathcal{K}\{\iota_1 x.c_1 \mid \iota_2 y.c_2\} \mid \tilde{\mu} x.H[\langle x \| E \rangle] \end{array}$$

Call-by-need reduction rules:

$$\begin{array}{llll} (\beta^\sharp_\mu) & \langle \mu\alpha.c \| E \rangle \to c[E/\alpha] & (\beta^\sharp_{\tilde{\mu}}) & \langle V \| \tilde{\mu} x.c \rangle \to c[V/x] \\ (\beta^\sharp_\to) & \big\langle \lambda[x \cdot \alpha].c \big\| V \cdot E \big\rangle \to c[V/x, E/\alpha] & (\beta^\sharp_\oplus) & \big\langle \iota_i V \big\| \mathcal{K}\{\iota_1 x_1.c_1 \mid \iota_2 x_2.c_2\} \big\rangle \to c_i[V/x_i] \end{array}$$

Figure 8: A call-by-need $\overline{\lambda}\mu\tilde{\mu}$ sub-calculus.

Sub-syntax for call-by-coneed evaluation:

$$\begin{array}{ll} c ::= \langle v \| e \rangle & H ::= \Box \mid \langle \mu\alpha.H \| e \rangle \\ v ::= V \mid \mu\alpha.c & V ::= x \mid \lambda[x \cdot \alpha].c \mid \iota_1 V \mid \iota_2 V \mid \mu\alpha.H[\langle V \| \alpha \rangle] \\ e ::= E \mid \tilde{\mu} x.c & E ::= \alpha \mid V \cdot E \mid \mathcal{K}\{\iota_1 x.c_1 \mid \iota_2 y.c_2\} \end{array}$$

Call-by-coneed reduction rules:

$$\begin{array}{llll} (\beta^\flat_\mu) & \langle \mu\alpha.c \| E \rangle \to c[E/\alpha] & (\beta^\flat_{\tilde{\mu}}) & \langle V \| \tilde{\mu} x.c \rangle \to c[V/x] \\ (\beta^\flat_\to) & \big\langle \lambda[x \cdot \alpha].c \big\| V \cdot E \big\rangle \to c[V/x, E/\alpha] & (\beta^\flat_\oplus) & \big\langle \iota_i V \big\| \mathcal{K}\{\iota_1 x_1.c_1 \mid \iota_2 x_2.c_2\} \big\rangle \to c_i[V/x_i] \end{array}$$

Figure 9: The dual of call-by-need in the $\overline{\lambda}\mu\tilde{\mu}$ calculus.

5.3. **The dual of call-by-need evaluation.** One of the strengths of the sequent calculus is the way it presents the duality between call-by-value and call-by-name evaluation [CH00, Wad03]. Since producers are dual to consumers, the dual can be found by just swapping terms with coterms and flipping the two sides of a command. This simple syntactic transformation makes it straightforward to discover the dual to classical call-by-need evaluation [AHS11], here called call-by-coneed. We only need to exchange the roles of values and covalues from the definition of call-by-need above, and to dualize heaps so that covariables are bound to delayed demands (a consumer which might not need its input yet), as shown in Figure 9. As with call-by-need, the reduction rules share the same form, but with a new definition for V and E.

Call-by-need evaluation can be seen as a more efficient approach to laziness compared with call-by-name, where the computation of named results are remembered and shared. Once control effects are involved, call-by-coneed evaluation can be seen as a more efficient approach to strictness compared with call-by-value, where the computation of labeled call sites are remembered and shared. For example, consider the following use of the control operator callcc in the λ-calculus:

$$\mathbf{let}\, x = \mathsf{callcc}(\lambda k.v)\, \mathbf{in}\, \mathbf{let}\, n = \mathit{fib}\ 30\, \mathbf{in}\, n + n$$

where the call to *fib* 30 represents an expensive computation. In call-by-value, the invocation of callcc is the first step, copying the context $\mathbf{let}\, x = \Box\, \mathbf{in}\, \mathbf{let}\, n = \mathit{fib}\ 30\, \mathbf{in}\, n + n$, and every time k is called *fib* 30 will be recomputed. However, in call-by-coneed, *fib* 30 is only computed once—the first time k is called with a value—after which the context is replaced with the normalized $\mathbf{let}\, x = \Box\, \mathbf{in}\, 1664080$ that can immediately return its result.

6. Compiling Call-by-need and its Dual

Previously in Section 3, we were able to encode the call-by-value and -name $\overline{\lambda}\mu\tilde{\mu}$-calculi into System L. How could we accomplish a similar thing for call-by-need and its dual? First of all, the main key to polarizing compilation is to use polarity shifts to mediate between the baked-in evaluation strategies of the basic connectives ($\to$ and $\oplus$) and the intended evaluation strategy of the source language. This means that System L on its own is not enough: before we can encode a language with a different evaluation strategy like call-by-need, we will need to have enough shifts to mediate between call-by-need and the call-by-value and -name strategies of the basic connectives.

But compilation should do more than just get the compiled program to type check; it should also exactly reflect the semantics of the source program in the target calculus. And on this point, we need to take great care with how the shifts interact with the semantics of evaluation strategies besides call-by-value and -name. Consider the first polarizing compilation we gave in Figure 6. Recall how the translation of a sum injection $(\iota_i V)^-$ ends up burying the sum inside of a λ-abstraction. This style of compilation is not acceptable for a call-by-need language, because it is sensitive to *sharing*: which results of sub-computations are computed only once and shared from then on.

In a call-by-need language, all named terms are shared, and moreover, the contents of a constructor like ι_i are *also* shared. In contrast, the body of a λ-abstraction is never shared in a call-by-need language, and is re-computed every time the function is called. This dichotomy does not fit with the compilation given in Figure 6 for $(\iota_i V)^-$ because the extra λ-abstraction surrounds the contents of the ι_i constructor and breaks sharing. And for the dual reason, the call-by-value compilation of $(V \cdot E)^+$ in Figure 6 does not work for call-by-coneed, because the extra λ-abstraction buries the function application destructor and breaks sharing of V and E.

However, in Section 4 we discussed how to get rid of these extra λ-abstraction barriers in the compiled code. The key was to use a different style of shift ($\Downarrow$ and $\Uparrow$) to convert the outside of the connective compared with the shifts ($\downarrow$ and $\uparrow$) used to wrap the sub-formulas of the connective. This alternate compilation avoids burying constructors and destructors inside of abstractions, and will therefore have the correct sharing semantics for an evaluation strategy like call-by-need.

So to compile another evaluation strategy into a language like System L, we will have to extend System L with the two different styles of shift to mediate between that evaluation strategy and call-by-value and -name. We can use the optimized compilation in Section 4 to guide our generalization. We need to use the thin shifts ($\downarrow$ and $\uparrow$) to convert a call-by-(co)need type to either call-by-value or call-by-name, so these shifts should be generalized on the kinds of types they can accept as inputs. And since we need to use the thick shifts ($\Downarrow$ and $\Uparrow$) to convert from a call-by-value or -name type built by one of $\to$ or $\oplus$ to a call-by-(co)need type, those shifts will should be generalized on the kinds of types they can give as results. In the end, we must extend the set of signs s to include symbols denoting call-by-need ($\sharp$) and call-by-coneed ($\flat$), and generalize both styles of the shift connectives as in Figure 10, where the unannotated shifts from before would now be written more explicitly as $\downarrow_-$, $\uparrow_+$, ${}_+\Downarrow$, and ${}_-\Uparrow$.

The generalized shifts let us give a generic polarizing compilation shown in Figure 11 for the $\overline{\lambda}\mu\tilde{\mu}$-calculi following *any* of these four evaluation strategies. For clarity and terseness, this generic compilation is written using nested (co)patterns. But the nested (co)patterns

$$s ::= + \mid - \mid \sharp \mid \flat \qquad A, B, C ::= X_s \mid A \to B \mid A \oplus B \mid \downarrow_s A \mid {}_s\Uparrow A \mid \uparrow_s A \mid {}_s\Downarrow A$$

$$\frac{}{X_s : s} \qquad \frac{A : s}{\downarrow_s A : +} \qquad \frac{A : +}{{}_s\Uparrow A : s} \qquad \frac{A : s}{\uparrow_s A : -} \qquad \frac{A : -}{{}_s\Downarrow A : s}$$

Figure 10: Generalized shifts to and from call-by-(co)need.

Generic polarizing compilation for $s \in \sharp, \flat, +, -$.

$$X^s \triangleq X_s \qquad (A \to B)^s \triangleq {}_s\Downarrow((\downarrow_s A^s) \to (\uparrow_s B^s)) \qquad (A \oplus B)^s \triangleq {}_s\Uparrow((\downarrow_s A^s) \oplus (\downarrow_s B^s))$$

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

$$x^s \triangleq x \qquad \alpha^s \triangleq x$$

$$(\mu\alpha.c)^s \triangleq \mu\alpha.(c^s) \qquad (\tilde{\mu}x.c)^s \triangleq \tilde{\mu}x.(c^s)$$

$$(\lambda[x \cdot \alpha].c)^s \triangleq \lambda[{}_s\Downarrow[[\downarrow_s x] \cdot [\uparrow_s \alpha]]].c^s \qquad (V \cdot E)^s \triangleq {}_s\Downarrow[[\downarrow_s V^s] \cdot [\uparrow_s E^s]]$$

$$(\iota_i V)^s \triangleq {}_s\Uparrow(\iota_i(\downarrow_s V^s)) \qquad (\mathcal{K}\{\iota_i x_i.c_i\})^s \triangleq \mathcal{K}\{ {}_s\Uparrow(\iota_i(\downarrow_s x_i)).c_i^s \}$$

Figure 11: Generic compilation of call-by-need and call-by-coneed using the four shifts.

can be mechanically flattened out into the simpler forms by matching one step at a time, analogous to the desugaring of pattern matching in a functional language, like so:

$$(\lambda[x \cdot \alpha].c)^s \triangleq \lambda_s\Downarrow\gamma.\Big\langle \lambda[y \cdot \beta].\, \langle y \big|_+ \big| \mathcal{K}\downarrow_s x.\langle \lambda\uparrow_s \alpha.c^s |_- | \beta \rangle \rangle \Big|_- \Big| \gamma \Big\rangle$$

$$(\mathcal{K}\{\iota_i x_i.c_i\})^s \triangleq \mathcal{K}\, {}_s\Uparrow z.\big\langle z \big|_+ \big| \mathcal{K}\{\iota_i y_i.\langle y_i |_+ | \mathcal{K}\downarrow_s x_i.c_i^s \rangle\} \big\rangle$$

Note that in this generic compilation for an evaluation strategy denoted by s, type variables X are just translated to type variables of the appropriate sign, X_s. Furthermore, every binary connective is expanded into a polarized connective interspersed with three shifts: two shifts (some combination of $\downarrow_s$ and $\uparrow_s$) to convert the two sub-formulas, and one shift (either ${}_s\Uparrow$ or ${}_s\Downarrow$) to convert the final type. In this sense, the compilation of $\oplus$ is exactly like the optimized call-by-name compilation given in Section 4, except generalized to an arbitrary s, because it already had the maximal number of shifts. In contrast, the compilation of $\to$ has one more shift than the optimized call-by-value compilation given in Section 4 because the type of the function argument is not known to be positive already.

7. The Dual Core Calculus: System $\mathcal{D}$

So far, we have seen two independent methods of integrating call-by-value and call-by-name features into a classical calculus: polarity which directly mixes the two, and call-by-(co)need evaluation which shares delayed computations. We then illustrated how we might reconcile these two, so that they can be used in tandem, by generalizing beyond a binary polarity to one which mixes sharing with the polarized connectives. Doing so allows us to embed each of call-by-value, -name, -need, and -coneed into a common core language. However, there remains some issues which still need to be resolved. How do we combine the semantics of all four evaluation strategies into a cohesive calculus? How do we extend our results beyond the

simple functions-and-sums testing ground to a wide variety of other types that commonly appear in programming languages?

We resolve these issues with the design of a core calculus, called $\mathcal{D}$,[3] which in addition to subsuming all of call-by-value, -name, -need, and -coneed evaluation it presents the following features:

- $\mathcal{D}$ only includes a fixed, finite set of basic types. As discussed in more detail later in Sections 9 and 10, these basic types are expressive enough to represent a wide range of types that a programmer might define and use in a programming language.
- Duality and symmetry plays a central role to our framework, and so we prefer a fully-dual presentation whenever possible. As such, $\mathcal{D}$ avoids the antisymmetric function type in favor of the symmetric *par* type $A \parr B$—inspired by Girard's linear logic [Gir87]—which is exactly dual to a tuple type. Unlike function types, where $A \to B$ is generally quite different from $B \to A$, par types have the pleasant property that $A \parr B$ is isomorphic to $B \parr A$.
- Modern programming languages with static types typically have some mechanism for type abstraction, so $\mathcal{D}$ includes two dual quantifiers which have a solid grounding in logic. The universal quantifier $\forall$ models parametric polymorphism [Gir72, Rey74, GTL89], and the existential quantifier $\exists$ models modules and abstract data types [Pie02, Har12]. Although these quantifiers are standard extensions, they impose some serious constraints when reasoning about the correctness of the encodings, as discussed later in Section 10.
- Since $\mathcal{D}$ is based on the classical sequent calculus, it already has a natural notion of control effect due to the duality between information flow and control flow. Therefore, our semantics and theory is robust in the face of effects like exceptions and recursion.

7.1. **Syntax.** The syntax of the $\mathcal{D}$ calculus, given in Figure 12, is an extension of System L that was presented in Section 3. Notice how commands $\langle v|A{:}s|e\rangle$ are not just marked with a sign s but also with the type A of both v and e. The only purpose of annotating the type A in the command is to aid in type checking, and it can otherwise be ignored. As such, we will just use the shorthand $\langle v|s|e\rangle$ when A is clear from context or irrelevant to the particular example. As additional shorthand, for clarity we will write binary connectives as infix (*e.g.,* $A \oplus B$ instead of $\oplus\ A\ B$) and the quantifiers with their binding (*e.g.,* $\forall X{:}s.A$ instead of $\forall_s(\lambda X{:}s.A)$).

Recall from Sections 2, 3 and 5 how substitutability was an essential notion for understanding the semantics of programs. Substitutability also plays a central role of our core calculus, which has a built-in notion of *discipline*, denoted by (s). A discipline is used to stipulate which terms (V_s) might replace variables and which coterms (E_s) might replace covariables. This is enough to distinguish the different semantics of the four evaluation strategies, which are marked by a different discipline symbol: call-by-value $(+)$, call-by-name $(-)$, call-by-need $(\sharp)$, and call-by-coneed $(\flat)$.

The four different disciplines are then defined through the respective V_s and E_s, which are essentially the same as the previous $\bar{\lambda}\mu\tilde{\mu}$-calculi from Sections 3 and 5. The only difference is that a heap H can contain now any mixture of both delayed call-by-need $(\sharp)$ variable bindings, and delayed call-by-coneed $(\flat)$ covariable bindings.

The $\mathcal{D}$ core calculus has several additional basic types than was discussed in Sections 2, 3 and 5, which can be understood in the following groups.

[3]So named because it is a fully $\mathcal{D}$ual calculus with many $\mathcal{D}$isciplines.

Commands c, terms v, coterms e, weak-head normal terms W, and forcing coterms F:

$$c ::= \langle v | A{:}s | e \rangle \qquad v ::= W \mid \mu\alpha.c \qquad e ::= F \mid \tilde{\mu}x.c$$

$$W ::= x \mid \lambda\{q_i.c_i \mid \ldots\} \mid \iota_1 W \mid \iota_2 W \mid (W, W') \mid () \mid \ominus F \mid A \mathbin{\mathfrak{V}} W \mid \downarrow_s V_s \mid {}_s{\Uparrow} W$$

$$F ::= \alpha \mid \mathcal{K}\{p_i.c_i \mid \ldots\} \mid \pi_1 F \mid \pi_2 F \mid [F, F'] \mid [] \mid \neg W \mid A \mathbin{@} F \mid \uparrow_s E_s \mid {}_s{\Downarrow} F$$

Patterns p and copatterns q:

$$p ::= \iota_1 x \mid \iota_2 x \mid (x, y) \mid () \mid \ominus\alpha \mid X \mathbin{\mathfrak{V}} y \mid \downarrow_s x \mid {}_s{\Uparrow} x$$

$$q ::= \pi_1 \alpha \mid \pi_2 \alpha \mid [\alpha, \beta] \mid [] \mid \neg x \mid X \mathbin{@} \alpha \mid \uparrow_s \alpha \mid {}_s{\Downarrow} \alpha$$

Discipline-specific values V_s, covalues E_s, and heap contexts H:

$$V_- ::= v \qquad V_+ ::= W \qquad V_\sharp ::= W \qquad V_\flat ::= W \mid \mu\alpha.H[\langle V_\flat | A{:}\flat | \alpha \rangle]$$

$$E_- ::= F \qquad E_+ ::= e \qquad E_\sharp ::= F \mid \tilde{\mu}x.H[\langle x | A{:}\sharp | E_\sharp \rangle] \qquad E_\flat ::= F$$

$$H ::= \Box \mid \langle v | A{:}\sharp | \tilde{\mu}x.H \rangle \mid \langle \mu\alpha.H | A{:}\flat | e \rangle$$

Types A, kinds k, disciplines s, and connectives F:

$$A, B, C ::= X \mid \mathsf{F} \mid \lambda X{:}k.A \mid A\ B \qquad k, l ::= s \mid k \to l \qquad r, s, t ::= + \mid - \mid \flat \mid \sharp$$

$$\mathsf{F}, \mathsf{G} ::= \& \mid \parr \mid \top \mid \bot \mid \oplus \mid \otimes \mid 0 \mid 1 \mid \neg \mid \ominus \mid \forall_s \mid \exists_s \mid \downarrow_s \mid \uparrow_s \mid {}_s{\Uparrow} \mid {}_s{\Downarrow}$$

Figure 12: Syntax of System $\mathcal{D}$: a core, fully dual, multi-discipline sequent calculus.

Positive data types. All positive data types hereditarily follow a call-by-value strategy. They have values which are built from other positive values with constructors and covalues which consume their input by matching on the possible constructions they might be given of the general form $\mathcal{K}\{p_1.c_1 \mid \cdots \mid p_n.c_n\}$ (we will drop the braces when there is only one branch). In addition to the sum type $A \oplus B$ from Section 2, we also have the tuple type $A \otimes B$ which contains pair values of the form (W_1, W_2) that can be matched with a covalue of the form $\mathcal{K}(x, y).c$ that unpacks the two components of the pair. We also include the nullary versions of sum and tuple types. The nullary tuple is the unit type 1, which contains a unit value $()$ and the matching covalue $\mathcal{K}().c$ that just waits to receive a $()$ before running c. The nullary sum is the empty type 0, which contains no constructed values, so that the matching covalue $\mathcal{K}\{\}$ has no branches, because there are no constructions it might receive.

Negative data types. All negative codata types hereditarily follow a call-by-name strategy. They have covalues which are built from other negative covalues with destructors and values which respond to their observer by matching on the possible destruction they might encounter of the general form $\lambda\{q_1.c_1 \mid \cdots \mid q_n.c_n\}$. Dual to the sum type, we have the product type $A \,\&\, B$ which contains projection covalues of the form $\pi_i F$ which is matched by a value of the form $\lambda\{\pi_1\alpha.c_1 \mid \pi_2\beta.c_2\}$ that will run the command c_1 if asked for its first component or c_2 if asked for its second component. Dual to the tuple type, we have the par type $A \parr B$ which contains the compound covalue $[F_1, F_2]$ that pairs up two other consumers and can be matched by a value of the form $\lambda[\alpha, \beta].c$ that unpacks the pair of consumers before running c. As before, we include the nullary versions of product and par types. The nullary product type is the trivial type $\top$, which contains no possible destructors, so that the matching value $\lambda\{\}$ has no branches since it will never need to respond to any request. The nullary par type is the absurd type $\bot$, which contains the terminal destructor $[]$ with the matching value $\lambda[].c$ that just waits for the empty signal $[]$ before running c.

Involutive negation. The two negation types can be thought of as two dual way for representing first-class continuations in a programming language. One way to formulate a continuation is by capturing the context as a first class value. This corresponds to the data type $\ominus A$ which packages up a covalue F as the value $\ominus F$, which can be later unpacked by pattern-matching in $\lambda\!\!\!\lambda \ominus\alpha.c$. Another way to formulate continuations is through functions that never return. This corresponds to the codata type $\neg A$ which has values of the form $\lambda\neg x.c$, which is analogous to a function abstraction that does not bind a return pointer, and covalues of the form $\neg W$, which is analogous to throwing a value to a continuation without waiting for a returned result. This analogy reverses the usual order of ideas by making continuations primitive, so that function types are instead derived like so:

$$A \to B = (\neg A) \parr B \qquad W \cdot F = [\neg W, F] \qquad \lambda[x \cdot \alpha].c = \lambda[\neg x, \alpha].c = \lambda[\beta, \alpha].\, \langle \lambda\neg x.c \| \beta \rangle$$

Also note that the two negation connectives belong to different disciplines and are, in fact, polarity inverting. The $\neg$ form of negation converts a positive type $A : +$ to the negative $\neg A : -$, and dually $\ominus$ converts a negative type $B : -$ to the positive $\ominus A : +$. The inversion between positive and negative corresponds to the inversion between input and output inherent in continuations, and is essential to making negation involutive [MM14]: $\ominus\neg A$ is isomorphic to A, and likewise $\neg \ominus B$ is isomorphic to B. This would not hold if we forced the negation connectives to preserve the same polarity of the given type as in [Zei09].

Type quantifiers. Lastly, we have the two quantifiers for abstracting over types in programs. The sequent calculus helps to show the duality of these two forms of type abstraction as a syntactic symmetry between producers and consumers. The universal quantifier $\forall$ is modeled as a negative codata type analogous to the quantifier of System F, and can quantify over any of the four kinds of types ($+$, $-$, $\sharp$, or $\flat$). Values of type $\forall X{:}s.A$ abstract over types of kind s similar to a function of the form $\lambda X \mathbin{@} \alpha.c$, which matches against a destructor of the form $B \mathbin{@} F$. Notice that this means the consumer chooses a private type B for its observation, which forces polymorphism on the value it interacts with. Dually, the existential quantifier $\exists$ is modeled as a positive data type, with exactly the reversed roles. Values of type $\exists X{:}s.A$ choose a private type used in another value which is not visible from the outside, and the two are packaged together as $B \mathbin{\text{Ⓥ}} W$. Consumers of existential packages must then abstract over the privately chosen type, of the form $\lambda\!\!\!\lambda X \mathbin{\text{Ⓥ}} y.c$ similar to pattern matching on tuples.

Shifts. So far, all the basic types above follow the standard polarized recipe. Our treatment of the shifts is where $\mathcal{D}$ really differs from other calculi. First, let's consider the shifts which are modeled as data types—that is, with constructors—which coerce other types into and out of the call-by-value world. The shift $\downarrow_s$ embeds an s-value of type $A : s$ to a positive data structure $\downarrow_s A : +$. The construction $\downarrow_s V_s$ can be thought of like a box containing the value V_s, where the notion of "value" depends on s. For example, if s is $-$, then V_s may be any term, but if it is $+$, then it must be some W. The matching form $\lambda\!\!\!\lambda \downarrow_s x.c$ unwraps this box and binds the value to x in c. In contrast, the shift ${}_s\Uparrow$ embeds a positive data structure $A : +$ as a value of type ${}_s\Uparrow A : s$ in any discipline. The construction ${}_s\Uparrow W$ can be thought of as returning the call-by-value answer W in some other context, and $\lambda\!\!\!\lambda {}_s\Uparrow x.c$ forces computations of type ${}_s\Uparrow A$ to bind this answer to x in c.

The shifts which are modeled as codata types–that is, with destructors—are dual to the above, and allow for coercions into and out of the call-by-name world. The shift $\uparrow_s$ embeds an s-covalue of type $A : s$, that is a substitutable consumer of As, into a negative

$$\oplus : + \to + \to + \quad \otimes : + \to + \to + \quad 0 : + \quad 1 : + \quad \ominus : - \to + \quad \exists_s : (s \to +) \to +$$
$$\mathbin{\&} : - \to - \to - \quad \parr : - \to - \to - \quad \top : - \quad \bot : - \quad \neg : + \to - \quad \forall_s : (s \to -) \to -$$
$$\downarrow_s : s \to + \quad \uparrow_s : s \to - \quad {}_s\Uparrow : + \to s \quad {}_s\Downarrow : - \to s$$

$$\frac{}{\Theta, X : k \vdash_{\mathcal{D}} X : k} \qquad \frac{\mathsf{F} : k}{\Theta \vdash_{\mathcal{D}} \mathsf{F} : k} \qquad \frac{\Theta, X : k \vdash_{\mathcal{D}} A : l}{\Theta \vdash_{\mathcal{D}} \lambda X{:}k.A : k \to l} \qquad \frac{\Theta \vdash_{\mathcal{D}} A : k \to l \quad \Theta \vdash_{\mathcal{D}} B : k}{\Theta \vdash_{\mathcal{D}} A\ B : l}$$

$$\frac{(\Theta \vdash_{\mathcal{D}} A : t) ... \quad (\Theta \vdash_{\mathcal{D}} B : r) ...}{(x : A : t ... \vdash^{\Theta}_{\mathcal{D}} \beta : B : r ...)\,\mathbf{ctx}}$$

Figure 13: Kinds of System $\mathcal{D}$ types.

observation of type $\uparrow_s A : -$. The destructor $\uparrow_s E_s$ can be thought of as recognizing that E_s is strict on its input, where the options for the covalue E_s depend on the discipline s. For example, if s is $+$, then any e is a strict covalue, but if s is $-$, then only a forcing F is strict. The matching $\lambda\uparrow_s \alpha.c$ is then a negative computation which binds α to this strict covalue. In contrast, the shift ${}_s\Downarrow$ embeds a negative observation of type $A : -$ as a covalue of type ${}_s\Downarrow A : s$ in other discipline. The observation ${}_s\Downarrow F$ can be thought of as an observation for forcing some thunk, which is represented by $\lambda_s\Downarrow\alpha.c$.

Intermezzo 7.1. Each of the connectives in Figure 12 has an intended discipline, which corresponds with an evaluation strategy. For example, the tuple type constructor takes two call-by-value types and returns another call-by-value type, *i.e.,* $\otimes : + \to + \to +$. Calculi in this vein sometimes embed some additional amount of well-formedness into the syntax of types themselves, rather than as an auxiliary check after the fact, especially when higher kinds of the form $k \to l$ are left out. The corresponding well-formed sub-syntax of non-higher-kinded $\mathcal{D}$ types (where type variables $X : s$ are explicitly annotated as X_s) is given by the following grammar, where we have a different syntax of types A_s for each discipline s:

$$A_+ ::= X_+ \mid {}_+\Uparrow A_+ \mid {}_+\Downarrow A_- \mid A_+ \oplus B_+ \mid A_+ \otimes B_+ \mid 0 \mid 1 \mid \ominus A_- \mid \exists X{:}s.A_+ \mid \downarrow_s A_s$$
$$A_- ::= X_- \mid {}_-\Uparrow A_+ \mid {}_-\Downarrow A_- \mid A_- \mathbin{\&} B_- \mid A_- \parr B_- \mid \top \mid \bot \mid \neg A_+ \mid \forall X{:}s.A_- \mid \uparrow_s A_s$$
$$A_\sharp ::= X_\sharp \mid {}_\sharp\Uparrow A_+ \mid {}_\sharp\Downarrow A_-$$
$$A_\flat ::= X_\flat \mid {}_\flat\Uparrow A_+ \mid {}_\flat\Downarrow A_-$$

A similar exercise can be done to sub-divide expressions (terms, coterms, *etc.*) according to discipline based on which evaluation strategy they follow. But the downside is an explosion of cases in the syntactic grammar.

7.2. **Type system.** The type system for the dual core calculus $\mathcal{D}$ is given in Figures 13 to 17. Figure 13 gives the kind system for checking that types are well-formed, which is standard for calculi based on System F with higher kinds. More specifically, the kinding rules have judgments of the form $\Theta \vdash_{\mathcal{D}} A : k$—where Θ assigns a kind to each free type variable X of A—and are equivalent to a simply-typed λ-calculus at the type level with variables, type constructors (for connectives), applications, and abstractions. The subscript $\mathcal{D}$ signifies that types are built using only the core connectives of System $\mathcal{D}$, as opposed to other sets of connectives that are allowed later on in Section 8. We can also check that a sequent is well-formed with $(\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)\,\mathbf{ctx}$ by checking that each type in Γ and Δ has

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} v : A \mid \Delta \quad \Theta \vdash_{\mathcal{D}} A : s \quad \Gamma \mid e : A \vdash^{\Theta}_{\mathcal{D}} \Delta}{\langle v | A{:}s | e \rangle : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)}\ \mathit{Cut}$$

$$\frac{}{\Gamma, x : A \vdash^{\Theta}_{\mathcal{D}} x : A \,;\, \Delta}\ \mathit{VR} \qquad \frac{}{\Gamma \,;\, \alpha : A \vdash^{\Theta}_{\mathcal{D}} \alpha : A, \Delta}\ \mathit{VL}$$

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

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} v : A \,;\, \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{D}} v : A \mid \Delta}\ \mathit{FR} \qquad \frac{\Gamma \,;\, e : A \vdash^{\Theta}_{\mathcal{D}} \Delta}{\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{D}} \Delta}\ \mathit{FL}$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} V_s : A \mid \Delta \quad \Theta \vdash_{\mathcal{D}} A : s}{\Gamma \vdash^{\Theta}_{\mathcal{D}} V_s : A \,;\, \Delta}\ \mathit{BR} \qquad \frac{\Theta \vdash_{\mathcal{D}} A : s \quad \Gamma \mid E_s : A \vdash^{\Theta}_{\mathcal{D}} \Delta}{\Gamma \,;\, E_s : A \vdash^{\Theta}_{\mathcal{D}} \Delta}\ \mathit{BL}$$

Figure 14: Typing rules of System $\mathcal{D}$ that apply to any type.

some kind s in the environment Θ. Finally, we will treat the standard $\alpha\beta\eta$ equality between same-kinded types as implicit for the purpose of type checking. Formally, if $A_1 =_{\alpha\beta\eta} A_2$ and $\Theta \vdash_{\mathcal{D}} A_i : k$, then A_1 and A_2 are considered the same type in the environment Θ.

The generic typing rules that apply to any type are given in Figure 14. The main departure from the previous polarized type system in Section 3 is that typing judgments using a stoup (;) have been generalized beyond syntactic forms W and F to the more semantic notions of value and covalue. That is, the judgment $\Gamma \vdash^{\Theta}_{\mathcal{D}} v : A \,;\, \Delta$ is well-formed when v is a value V_s and A is an s-type such that $\Theta \vdash_{\mathcal{D}} A : s$ is derivable, and similarly for the dual judgment. By relaxing syntactic stipulations on the stoup, we can accommodate call-by-need (having a notion of covalue that is bigger than F and smaller than e) and call-by-coneed (with values between W and v). To formalize this idea, we add the *BR* and *BL* rules—opposite in direction to *FR* and *FL*—that *only* apply to values and covalues as defined by the discipline of their type. By the analogy in Intermezzo 3.1, this gives a systematic approach to the connection between evaluation and focusing that applies to more strategies than just call-by-value and call-by-name.

Note that type checking is still decidable even though (co)variables binders are not annotated with their type. This is possible because of the sequent calculus' *sub-formula property* [Gen35]—every type appearing in a premise of an inference rule appears somewhere in the conclusion of that rule. The only exception to the sub-formula property is the *Cut* rule that introduces arbitrary new types in the premises. The new type introduced by a *Cut* is made manifest in the syntax of expressions as the only necessary typing annotation, so that there is never any need to "guess" or infer during type checking. The ability to avoid inference altogether is an interesting property of the sequent calculus not shared by the λ-calculus. Effectively, every elimination rule of the λ-calculus is "hiding" an implicit cut, which forces us to reconstruct the type of the implicit cut during checking. The sub-formula property also aids in separating typing from checking that judgments are well-formed by just checking that the type of a *Cut* makes sense in the current environment: if the conclusion of an inference rule is well-formed (*i.e.,* $(\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)\,\mathbf{ctx}$), then so too are the premises.

Finally, we have the typing rules for specific connectives: positive types are given in Figure 15, negative types in Figure 16, and the shifts in Figure 17. The typing rules for positive and negative types are standard for a sequent-calculus presentation like System L

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} W : A_i \; ; \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \iota_i W : A_1 \oplus A_2 \; ; \Delta} \oplus R_i \qquad \frac{c_1 : (\Gamma, x : A \vdash^{\Theta}_{\mathcal{D}} \Delta) \quad c_2 : (\Gamma, y : B \vdash^{\Theta}_{\mathcal{D}} \Delta)}{\Gamma \; ; \curlywedge\{\iota_1 x.c_1 | \iota_2 y.c_2\} : A \oplus B \vdash^{\Theta}_{\mathcal{D}} \Delta} \oplus L$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} W_1 : A \; ; \Delta \quad \Gamma \vdash^{\Theta}_{\mathcal{D}} W_2 : B \; ; \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{D}} (W_1, W_2) : A \otimes B \; ; \Delta} \otimes R \qquad \frac{c : (\Gamma, x : A, y : B \vdash^{\Theta}_{\mathcal{D}} \Delta)}{\Gamma \; ; \curlywedge(x, y).c : A \otimes B \vdash^{\Theta}_{\mathcal{D}} \Delta} \otimes L$$

$$\text{no } 0R \text{ rules} \qquad \frac{}{\Gamma \; ; \curlywedge\{\} : 0 \vdash^{\Theta}_{\mathcal{D}} \Delta} 0L \qquad \frac{}{\Gamma \vdash^{\Theta}_{\mathcal{D}} () : 1 \; ; \Delta} 1R \qquad \frac{c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)}{\Gamma \; ; \curlywedge().c : 1 \vdash^{\Theta}_{\mathcal{D}} \Delta} 1L$$

$$\frac{\Gamma \; ; F : A \vdash^{\Theta}_{\mathcal{D}} \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \ominus F : \ominus A \; ; \Delta} \ominus R \qquad \frac{c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \alpha : A, \Delta)}{\Gamma \; ; \curlywedge \ominus \alpha.c : \ominus A \vdash^{\Theta}_{\mathcal{D}} \Delta} \ominus L$$

$$\frac{\Theta \vdash_{\mathcal{D}} B : s \quad \Gamma \vdash^{\Theta}_{\mathcal{D}} W : A \; B \; ; \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{D}} B \mathbin{\textcircled{\scriptsize D}} W : \exists_s A \; ; \Delta} \exists R \qquad \frac{c : (\Gamma, y : A \; X \vdash^{\Theta, X : s}_{\mathcal{D}} \Delta)}{\Gamma \; ; \curlywedge(X \mathbin{\textcircled{\scriptsize D}} y).c : \exists_s A \vdash^{\Theta}_{\mathcal{D}} \Delta} \exists L$$

Figure 15: Typing rules of the System $\mathcal{D}$ positive connectives.

$$\frac{\Gamma \; ; F : A_i \vdash^{\Theta}_{\mathcal{D}} \Delta}{\Gamma \; ; \pi_i F : A_1 \mathbin{\&} A_2 \vdash^{\Theta}_{\mathcal{D}} \Delta} \mathbin{\&} L_i \qquad \frac{c_1 : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \alpha : A, \Delta) \quad c_2 : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \beta : B, \Delta)}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \lambda\{\pi_1 \alpha.c_1 \mid \pi_2 \beta.c_2\} : A \mathbin{\&} B \; ; \Delta} \mathbin{\&} R$$

$$\frac{\Gamma \; ; F_1 : A \vdash^{\Theta}_{\mathcal{D}} \Delta \quad \Gamma \; ; F_2 : B \vdash^{\Theta}_{\mathcal{D}} \Delta}{\Gamma \; ; [F_1, F_2] : A \parr B \vdash^{\Theta}_{\mathcal{D}} \Delta} \parr L \qquad \frac{c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \alpha : A, \beta : B, \Delta)}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \lambda[\alpha, \beta].c : A \parr B \; ; \Delta} \parr R$$

$$\frac{}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \lambda\{\} : \top \; ; \Delta} \top R \qquad \text{no } \top L \text{ rules} \qquad \frac{}{\Gamma \; ; [] : \bot \vdash^{\Theta}_{\mathcal{D}} \Delta} \bot L \qquad \frac{c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \lambda[].c : \bot \; ; \Delta} \bot R$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} W : A \; ; \Delta}{\Gamma \; ; \neg W : \neg A \vdash^{\Theta}_{\mathcal{D}} \Delta} \neg L \qquad \frac{c : (\Gamma, x : A \vdash^{\Theta}_{\mathcal{D}} \Delta)}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \lambda \neg x.c : \neg A \; ; \Delta} \neg R$$

$$\frac{\Theta \vdash_{\mathcal{D}} B : s \quad \Gamma \; ; F : A \; B \vdash^{\Theta}_{\mathcal{D}} \Delta}{\Gamma \; ; B \mathbin{@} F : \forall_s A \vdash^{\Theta}_{\mathcal{D}} \Delta} \forall L \qquad \frac{c : (\Gamma \vdash^{\Theta, X : s}_{\mathcal{D}} \alpha : A \; X, \Delta)}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \lambda[X \mathbin{@} \alpha].c : \forall_s A \; ; \Delta} \forall R$$

Figure 16: Typing rules of the System $\mathcal{D}$ negative connectives.

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} V_s : A \; ; \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \downarrow_s V_s : \downarrow_s A \; ; \Delta} \downarrow R \qquad \frac{c : (\Gamma, x : A \vdash^{\Theta}_{\mathcal{D}} \Delta)}{\Gamma \; ; \curlywedge \downarrow_s x.c : \downarrow_s A \vdash^{\Theta}_{\mathcal{D}} \Delta} \downarrow L$$

$$\frac{c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \alpha : A, \Delta)}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \lambda_s \Downarrow \alpha.c : {}_s\Downarrow A \; ; \Delta} \Downarrow R \qquad \frac{\Gamma \; ; F : A \vdash^{\Theta}_{\mathcal{D}} \Delta}{\Gamma \; ; {}_s\Downarrow F : {}_s\Downarrow A \vdash^{\Theta}_{\mathcal{D}} \Delta} \Downarrow L$$

$$\frac{c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \alpha : A, \Delta)}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \lambda \uparrow_s \alpha.c : \uparrow_s A \; ; \Delta} \uparrow R \qquad \frac{\Gamma \; ; E_s : A \vdash^{\Theta}_{\mathcal{D}} \Delta}{\Gamma \; ; \uparrow_s E_s : \uparrow_s A \vdash^{\Theta}_{\mathcal{D}} \Delta} \uparrow L$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} W : A \; ; \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{D}} {}_s\Uparrow W : {}_s\Uparrow A \; ; \Delta} \Uparrow R \qquad \frac{c : (\Gamma, x : A \vdash^{\Theta}_{\mathcal{D}} \Delta)}{\Gamma \; ; \curlywedge {}_s\Uparrow x.c : {}_s\Uparrow A \vdash^{\Theta}_{\mathcal{D}} \Delta} \Uparrow L$$

Figure 17: Typing rules of the System $\mathcal{D}$ polarity shift connectives.

Disciplined substitutions (ρ):

$$\rho ::= A/X{:}k..., \ V_t/x{::}t..., \ E_r/\alpha{::}r...$$

Reduction rules:

$$(\beta_\mu) \quad \langle \mu\alpha.c | A{:}s | E_s \rangle \to c[E_s/\alpha{::}s] \qquad (\beta_{\tilde\mu}) \quad \langle V_s | A{:}s | \tilde\mu x.c \rangle \to c[V_s/x{::}s]$$

$$(\eta_\mu) \quad \mu\alpha.\langle v | A{:}s | \alpha \rangle \to v \quad (\alpha \notin FV(v)) \qquad (\eta_{\tilde\mu}) \quad \tilde\mu x.\langle x | A{:}s | e \rangle \to e \quad (x \notin FV(e))$$

$$(\beta_q) \quad \big\langle \lambda\{q_i.c_i..\} \big| A{:}s \big| q[\rho] \big\rangle \to c_i[\rho] \quad (q{=}q_i) \qquad (\beta_p) \quad \big\langle p[\rho] \big| A{:}s \big| \curlywedge\{p_i.c_i..\} \big\rangle \to c_i[\rho] \quad (p{=}p_i)$$

$$(\eta_q) \quad \lambda\{q.\langle x | A{:}s | q \rangle ...\} \to x \quad (x \notin FV(q)) \qquad (\eta_p) \quad \curlywedge\{p.\langle p | A{:}s | \alpha \rangle ...\} \to \alpha \quad (\alpha \notin FV(p))$$

$$(\chi^\sharp) \quad \big\langle \mu\alpha.\langle v | B{:}\sharp | \tilde\mu y.c \rangle \big| A{:}\sharp \big| e \big\rangle \to \big\langle v \big| B{:}\sharp \big| \tilde\mu y.\langle \mu\alpha.c | A{:}\sharp | e \rangle \big\rangle \quad (x \notin FV(e), \alpha \notin FV(v))$$

$$(\chi^\flat) \quad \big\langle v \big| A{:}\flat \big| \tilde\mu x.\langle \mu\beta.c | B{:}\flat | e \rangle \big\rangle \to \big\langle \mu\beta.\langle v | A{:}\flat | \tilde\mu x.c \rangle \big| B{:}\flat \big| e \big\rangle \quad (x \notin FV(e), \alpha \notin FV(v))$$

Typed equality relation:

$$\frac{c : (\Gamma \vdash^\Theta_{\mathcal{D}} \Delta) \quad c \to c'}{c = c' : (\Gamma \vdash^\Theta_{\mathcal{D}} \Delta)} \qquad \frac{\Gamma \vdash^\Theta_{\mathcal{D}} v : A \mid \Delta \quad v \to v'}{\Gamma \vdash^\Theta_{\mathcal{D}} v = v' : A \mid \Delta} \qquad \frac{\Gamma \vdash^\Theta_{\mathcal{D}} e : A \mid \Delta \quad e \to e'}{\Gamma \mid e = e' : A \vdash^\Theta_{\mathcal{D}} \Delta}$$

plus inference rules for compatibility, reflexivity, symmetry, transitivity of equality

Figure 18: Equational theory for System $\mathcal{D}$.

[MM13], and the typing rules for shifts are the same as in Section 3. Note how the more systematic approach to focusing shows the difference between the two different styles of shifts, the data shifts $\downarrow_s$ and ${}_s\Uparrow$ both lose focus on the left (because their premise is a command), whereas the dual codata shifts $\uparrow_s$ and ${}_s\Downarrow$ both lose focus on the right.

7.3. **Equational theory.** In order to reason about the behavior of expressions, we give an equational theory of the $\mathcal{D}$ calculus in Figure 18. Notice that it has the same β_μ and $\beta_{\tilde\mu}$ rules as System L from Section 3: the discipline sign s in the command is used to decide which notion of value or covalue is allowed. To make sure that this discipline is being respected during a substitution, we use the restricted form $[V_s/x{::}s]$ and $[E_s/\alpha{::}s]$ such that the syntactic sign s is matched by the syntactic category V_s or E_s. In other words, each discipline has its own notion of valid substitutions, and the discipline of a bound variable tells us what values or covalues might replace it. In contrast, the type annotating a command has no impact on the reductions it may take, and so they can be ignored.

The various β rules for specific data and codata types (like sum types and shifts) are all summarized by the general β_p and β_q rules, which perform a pattern or copattern match. The β_p rule says that if a constructed value can be decomposed as a pattern p and substitution ρ, then the matching branch of the covalue can be taken. For example, we would have the following instance of the rule for tuple patterns (ignoring the type annotation):

$$\big\langle (W_1, W_2) \big|_+ \big| \curlywedge(x,y).c \big\rangle = \big\langle (x,y)[W_1/x{::}_+, W_2/y{::}_+] \big|_+ \big| \curlywedge(x,y).c \big\rangle \to_{\beta_p} c[W_1/x{::}_+, W_2/y{::}_+]$$

The β_q rule copattern matches against a destruction, like in the following specific instance of the rule for product copatterns:

$$\big\langle \lambda\{\pi_1\alpha.c_1 \mid \pi_2\beta.c_2\} \big|_- \big| \pi_2 F \big\rangle = \big\langle \lambda\{\pi_1\alpha.c_1 \mid \pi_2\beta.c_2\} \big|_- \big| (\pi_2\beta)[F/\beta{::}_-] \big\rangle \to_{\beta_q} c_2[F/\beta{::}_-]$$

We also have reductions for recognizing extensionality, which further brings out the duality between data and codata. The η_μ and $\eta_{\tilde\mu}$ rules are the counterpart of β_μ and $\beta_{\tilde\mu}$, and

eliminate a redundant μ- or $\tilde{\mu}$-abstraction that introduces a (co)variable just to immediately use it exactly once, before forgetting it. Likewise, the η_q and η_p rules perform the same kind of reduction but for redundant (co)pattern matching instead of a plain (co)variable. Note that the important difference between η_μ and η_q is that η_μ applies to any underlying term v, whereas η_q does not. This is because the more general case is not sound for a codata type like ${}_+\Downarrow A$: there can be a difference between a term v which *computes* a thunk and $\lambda\{{}_+\Downarrow\alpha.\langle v|_+|{}_+\Downarrow\alpha\rangle\}$ which *is* a thunk. For example, consider the term $\mu\alpha.c_0 : {}_+\Downarrow A$ which throws away its observer because α does not appear in c_0. This term's η_q-expansion could give a different result when run with the coterm $\tilde{\mu}z.c_1$ which throws away its input because z does not appear in c_1, like so:

$$\langle \mu\alpha.c_0|_+|\tilde{\mu}z.c_1\rangle \mapsto_{\beta^+_\mu} c_0$$
$$\left\langle \lambda\left\{{}_+\Downarrow\alpha'.\left\langle \mu\alpha.c_0\middle|_+\middle|{}_+\Downarrow\alpha'\right\rangle\right\}\middle|_+\middle|\tilde{\mu}z.c_1\right\rangle \mapsto_{\beta^+_\mu} c_1$$

Therefore, the η_q rule is restricted to only apply to a variable x, which semantically stands for any value of the appropriate discipline, thereby avoiding the above counterexample. But for negative codata types, like products, we can derive the full η law as follows [DA14b]:

$$\begin{aligned}
v &\leftarrow_{\eta_\mu} \mu\alpha.\langle v|A\&B:_-|\alpha\rangle \\
&\leftarrow_{\beta^-_\mu} \mu\alpha.\left\langle v\middle|A\&B:_-\middle|\tilde{\mu}x.\langle x|_-|\alpha\rangle\right\rangle \\
&\leftarrow_{\eta_q} \mu\alpha.\left\langle v\middle|A\&B:_-\middle|\tilde{\mu}x.\left\langle \lambda\{\pi_1\beta_1.\langle x|_-|\pi_1\beta_1\rangle \mid \pi_2\beta_2.\langle x|_-|\pi_1\beta_2\rangle\}\middle|_-\middle|\alpha\right\rangle\right\rangle \\
&\rightarrow_{\beta^-_\mu} \mu\alpha.\left\langle \lambda\{\pi_1\beta_1.\langle v|_-|\pi_1\beta_1\rangle \mid \pi_2\beta_2.\langle v|_-|\pi_1\beta_2\rangle\}\middle|_-\middle|\alpha\right\rangle \\
&\rightarrow_{\eta_\mu} \lambda\{\pi_1\beta_1.\langle v|_-|\pi_1\beta_1\rangle \mid \pi_2\beta_2.\langle v|_-|\pi_1\beta_2\rangle\}
\end{aligned}$$

Dually, $\eta_{\tilde{\mu}}$ applies to any underlying coterm but η_p does not because it would not be sound for a data type like ${}_-\Uparrow A$: there can be a difference between a coterm e that *computes* a strict observation and $\lambda\{{}_-\Uparrow x.\langle {}_-\Uparrow x|{}_-\Uparrow A:_-|e\rangle\}$ which *is* a strict observation. For the same reason as before, the η_p rule is restricted to covariables, which still derives the full η law for positive data types, like the sum types in Section 3.

Finally, we have the re-association rules χ, which swap the bindings of a variable and covariable both of kind $\sharp$ or $\flat$. Observe how on both the left- and right-hand sides of $\chi^\sharp$ rule, a call-by-need evaluation order will first evaluate the coterm e, and if e reduces to covalue $E_\sharp$, then the two sides step to the same command $\left\langle v\middle|B:\sharp\middle|\tilde{\mu}y.c[E_\sharp/\alpha::\sharp]\right\rangle$ via a $\beta^\sharp_\mu$ reduction. Dually, in both the left- and right-hand sides of the $\chi^\flat$ rule, a call-by-coneed evaluation order will first evaluate v, and if v reduces to a value then both sides step to the same command via a $\beta^\flat_{\tilde{\mu}}$ reduction. Note that instances of these rules for a pair of call-by-value ($+$) or call-by-name ($-$) bindings are also sound, and can be derived from the β^+_μ and $\beta^-_{\tilde{\mu}}$ rules, respectively, like so:

$$\left\langle \mu\alpha.\langle v|_+|\tilde{\mu}y.c\rangle\middle|_+\middle|e\right\rangle \rightarrow_{\beta^+_\mu} \left\langle v\middle|_+\middle|\tilde{\mu}y.c[e/\alpha::_+]\right\rangle \leftarrow_{\beta^+_\mu} \left\langle v\middle|_+\middle|\tilde{\mu}y.\langle \mu\alpha.c|_+|e\rangle\right\rangle$$
$$\left\langle v\middle|_-\middle|\tilde{\mu}x.\langle \mu\beta.c|_-|e\rangle\right\rangle \rightarrow_{\beta-} \left\langle \mu\beta.c[v/x::_+]\middle|_-\middle|e\right\rangle \leftarrow_{\beta-} \left\langle \mu\beta.\langle v|_-|\tilde{\mu}x.c\rangle\middle|_-\middle|e\right\rangle$$

However, trying to reassociate arbitrary variable and covariable bindings in this way is not sound [MM13]; in particular, reassociating a call-by-value coterm and a call-by-name term binding can break confluence, as in (where δ and z are unused):

$$c_0 \leftarrow_{\beta^+_\mu} \left\langle \mu\delta.c_0\middle|_+\middle|\tilde{\mu}x.\langle \mu\alpha.c|_-|\tilde{\mu}z.c_1\rangle\right\rangle \neq_\chi \left\langle \mu\alpha.\langle \mu\delta.c_0|_+|\tilde{\mu}x.c\rangle\middle|_-\middle|\tilde{\mu}z.c_1\right\rangle \rightarrow_{\beta^-_{\tilde{\mu}}} c_1$$

7.4. **Operational semantics.** To give an operational semantics to System $\mathcal{D}$, we only need to pick a standard reduction from the rules of Figure 18. The goal is to define a standard reduction relation on commands, $c \mapsto c'$, with the following properties:

Property 7.2 (Standard reduction)**.**

a) *Determinism*: If $c_1 \leftarrow\!\shortmid\, c \mapsto c_2$ then c_1 and c_2 must be the same command.
b) *Compatibility with heaps*: If $c \mapsto c'$ then $H[c] \mapsto H[c']$ for any heap context H.
c) *Closure under substitution*: If $c \mapsto c'$ then $c[\rho] \mapsto c'[\rho]$ for any well-disciplined substitution ρ matching the free (co)variables of c.

As a first cut, we can just keep the essential reduction rules for computation (the β rules) and exclude the additional reductions that are only used for reasoning about program equivalence (the η and χ rules). Allowing reduction only on the top-level of a command is not enough, due to the delayed bindings caused by call-by-(co)need evaluation (as in Section 5), so we should allow reduction inside of heaps.

However, this immediately causes a problem with determinism. First of all, reduction inside heaps opens up the choice between substituting an unneeded binding early or not. For example, we could have two possible reductions in the command

$$\langle V_\sharp | \sharp | \tilde{\mu}x.\, \langle (W_1, W_2) | + | \curlywedge(y_1, y_2).c \rangle \rangle \to_{\beta^\sharp_{\tilde{\mu}}} \langle (W_1, W_2) | + | \curlywedge(x, y).c \rangle\, [V_\sharp / x :: \sharp]$$

$$\langle V_\sharp | \sharp | \tilde{\mu}x.\, \langle (W_1, W_2) | + | \curlywedge(y_1, y_2).c \rangle \rangle \to_{\beta_p} \langle V_\sharp | \sharp | \tilde{\mu}x.c[W_1/y_1 ::_+, W_2/y_2 ::_+] \rangle$$

To avoid this ambiguity, we will only fire a $\beta^\sharp_{\tilde{\mu}}$ reduction when the $\tilde{\mu}$-abstraction cannot take a step itself, *i.e.,* it is a covalue. Secondly, forcing a chain of variable bindings causes another ambiguity for which variable should be substituted first. Consider the command

$$\langle x | \sharp | \tilde{\mu}y.\, \langle y | \sharp | \tilde{\mu}z.\, \langle z | \sharp | \alpha \rangle \rangle \rangle$$

where z is forced by α, which forces evaluation of y, and again forces evaluation of x. Because of this chain of demand, each of α, $\tilde{\mu}z.\, \langle z | \sharp | \alpha \rangle$, *and* $\tilde{\mu}y.\, \langle y | \sharp | \tilde{\mu}z.\, \langle z | \sharp | \alpha \rangle \rangle$ are all call-by-need covalues. So which variable substitution should occur first? If we say the outermost one, then standard reduction is not compatible with heaps (that could add another outer-most binding). If we say the innermost one, then standard reduction is not closed under substitution (that could replace α and add another inner-most binding). But at the end of the day this decision doesn't matter; we still need to know the value of the last variable x to make meaningful progress. So instead we rule out this scenario altogether: $\beta^\sharp_{\tilde{\mu}}$ reduction will not fire for variables, and $\beta^\flat_\mu$ reduction will not fire for covariables.

With these observations in mind, the operational semantics of $\mathcal{D}$ is given in Figure 19. The standard reduction relation can reduce any command inside of a heap context and includes the (co)pattern-matching rules β_p and β_q. For reducing μ- and $\tilde{\mu}$-abstractions, we use a special restricted rule $\phi^\sharp_{\tilde{\mu}}$ for call-by-need $\tilde{\mu}$s and $\phi^\flat_\mu$ for call-by-coneed μs. In all other cases, we just use the unaltered β_μ and $\beta_{\tilde{\mu}}$ rules from the equational theory in Figure 18. Notice that this definition of standard reduction satisfies each of the properties in Property 7.2. It is compatible with heaps by definition. And both determinism and closure under substitution follow by induction on the syntax of commands (using the fact that c has no standard reduction if $\tilde{\mu}x.c$ is a $\sharp$-covalue, or dually if $\mu\alpha.c$ is a $\flat$-value).

So now that we know how to run commands, how do we know when they're done? Since we will need to be able to run *open* commands which may have free variables and covariables due to (co)call-by-need evaluation, the idea of *needed (co)variables* will become

Standard reduction rules:

$$(\beta_\mu^{\pm\sharp}) \quad \langle \mu\alpha.c | A{:}s | E_s \rangle \mapsto c[E_s/\alpha{::}s] \qquad (s \in \{+, -, \sharp\})$$

$$(\beta_{\tilde{\mu}}^{\pm\flat}) \quad \langle V_s | A{:}s | \tilde{\mu}x.c \rangle \mapsto c[V_s/x{::}s] \qquad (s \in \{+, -, \flat\})$$

$$(\phi_\mu^\flat) \quad \langle \mu\alpha.c | A{:}\flat | E_\flat \rangle \mapsto c[E_\flat/\alpha{::}\flat] \qquad (\mu\alpha.c \in V_\flat,\ E_\flat \neq \beta)$$

$$(\phi_{\tilde{\mu}}^\sharp) \quad \langle V_\sharp | A{:}\sharp | \tilde{\mu}x.c \rangle \mapsto c[V_\sharp/x{::}\sharp] \qquad (\tilde{\mu}x.c \in E_\sharp, V_\sharp \neq y)$$

$$(\beta_p) \quad \big\langle p[\rho] \big| A{:}s \big| \curlywedge\{p_i.c_i.\dot{.}\} \big\rangle \mapsto c_i[\rho] \qquad (p = p_i)$$

$$(\beta_q) \quad \big\langle \lambda\{q_i.c_i.\dot{.}\} \big| A{:}s \big| q[\rho] \big\rangle \mapsto c_i[\rho] \qquad (q = q_i)$$

Reduction inside heap contexts:

$$\frac{c \mapsto c'}{H[c] \mapsto H[c']}$$

Figure 19: Operational semantics of System $\mathcal{D}$.

the cornerstone of a command's status by expressing when the value of a variable or covariable needs to be supplied for computation to continue. The key intuition about needed (co)variables, as defined above, is that (1) they are only needed in commands that can no longer take a standard reduction step, and (2) they are always found in the eye of a heap context H which does not bind them.

Definition 7.3 (Need)**.** The set of *needed* (co)variables of a command, written $\mathrm{NV}(c)$, is the smallest set such that:

a) $x \in \mathrm{NV}(c)$ if and only if $c \not\mapsto$ and $c = H[\langle x | A{:}s | E_s \rangle]$ such that x is not bound by H.
b) $\alpha \in \mathrm{NV}(c)$ if and only if $c \not\mapsto$ and $c = H[\langle V_s | A{:}s | \alpha \rangle]$ such that α is not bound by H.

The status of a command is then defined by its ability to take another step, or else which (co)variables it needs to continue stepping. This lets us formulate a notion of type safety that is both perfectly symmetric (with respect to free inputs and outputs) and open enough to capture call-by-coneed evaluation. In particular, a command is finished when supplying the (co)value for a (co)variable might spur it on to continue computation, but is stuck when no such substitution can ever restart computation. For example, the command $\big\langle \iota_1 x \big| + \big| \lambda\{\} \big\rangle$ is forever stuck.

Definition 7.4 (Status)**.** A command is *finished* when $\mathrm{NV}(c)$ is non-empty, and *stuck* when $c \not\mapsto$ and $\mathrm{NV}(c)$ is empty.

Theorem 7.5 (Type safety)**.** *a)* Progress*: If* $c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)$*, then* c *is not stuck.*
b) Preservation*: If* $c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)$ *and* $c \mapsto c'$ *then* $c' : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)$*.*

8. Enriching the Core: System $\mathcal{CD}$

Our first stated goal of the dual core calculus $\mathcal{D}$ is to be able to encode a wide range of types that a programmer might define and use in a programming language. Evaluating if this goal has been met is more subtle than just checking that enough well-typed programs can be written. For example, it is well-known that unary function types $(A \to B)$ can encode binary and higher-arity functions through currying, like $A \to (B \to C)$. However, in the

call-by-value λ-calculus, the curried function type $A \to (B \to C)$ is not the same thing as the binary function type $(A \otimes B) \to C$: there is no function of type $(A \otimes B) \to C$ that corresponds to the function $\lambda x.\Omega : A \to (B \to C)$ (where Ω is a term that loops forever) that diverges after being partially applied to one argument. Similarly, nesting sum types like $A \oplus (B \oplus C)$ as a way to encode a ternary choice has the analogous issue in the call-by-name λ-calculus: there is no term of a ternary choice type corresponding to $\iota_2\Omega$. Instead, we should ask for more than just well-typed programs: the encodings should be between types that are truly isomorphic to one another.

So how can we properly evaluate what class of types are representable by the $\mathcal{D}$ calculus? We will define an extension of $\mathcal{D}$, called $\mathcal{CD}$,[4] that is a higher-level language that is rich with many types, meeting the following objectives:

- Formalizing a fully-dual mechanism for programmatically declaring new data and codata types. To ensure generality, there will be no restrictions on the number of constructors or destructors, inputs or outputs, and disciplines involved. Declared (co)data types will also be able to introduce publically and privately quantified types, to give a general form of type abstraction.
- Removing the restrictions on constructor and destructor applications. In the extended $\mathcal{D}$ calculus, they can be applied to any terms and coterms, not just values and covalues.

On the one hand, the extension will clearly subsume $\mathcal{D}$ (that is, it is a conservative extension), so that nothing is lost. But on the other hand, we will show that the extended language $\mathcal{CD}$ can still be faithfully encoded back into the $\mathcal{D}$ core calculus via macro expansions, so that nothing of importance is gained. This way, the $\mathcal{CD}$ calculus reveals the expressive power of the $\mathcal{D}$ core calculus.

8.1. **Extending $\mathcal{D}$ with fully dual (co)data types.** The syntax for the $\mathcal{CD}$ calculus is given in Figure 20. Notice how, without the restrictions of the core calculus from Figure 12, the syntax is much simpler. Instead of the specific patterns and copatterns, we just have a general class of constructors (denoted by K) and destructors (denoted by O) names to choose from, which can be applied to any number of types, terms, and coterms. This unifies all the different cases into just one case for a data construction and one case for a codata destruction. Likewise at the type-level, instead of the list of specific connectives, we just have a class of connective names (denoted by F or G). These connective names are given meaning by a data or codata declaration, which must be non-recursive.[5]

In general, a data type declaration is similar to an algebraic data type (ADT) declaration from a functional language, and has the form:

$$\begin{aligned}&\textbf{data}\,\mathsf{F}(X{:}k)... : s\,\textbf{where}\\ &\quad \mathsf{K}_1 : (A_1 : t_1... \vdash^{Y_1:s_1...} \mathsf{F}\,X... \mid B_1 : r_1...)\\ &\quad ...\\ &\quad \mathsf{K}_n : (A_n : t_n... \vdash^{Y_n:s_n...} \mathsf{F}\,X... \mid B_n : r_n...)\end{aligned}$$

This declaration says that the connective F is parameterized by several types (named X, each of kind k, respectively) and returns a type of the discipline s. There are n different

[4] So named because it is a calculus of both $\mathcal{D}$ata and $\mathcal{C}o\mathcal{D}$ata.

[5] The requirement that codata declarations be non-recursive is so that we will be able to fully encode them via macro expansions, which only terminates for non-recursive declarations.

Commands c, terms v, coterms e, patterns p, and copatterns q

$$c ::= \langle v | A{:}s | e \rangle$$
$$v ::= x \mid \mu\alpha.c \mid \mathsf{K}\, B...e...v... \mid \lambda\{q.c \mid ...\} \qquad p ::= \mathsf{K}\, X...\alpha...y...$$
$$e ::= \alpha \mid \tilde{\mu} x.c \mid \mathsf{O}\, B...v...e... \mid \lambda\{p.c \mid ...\} \qquad q ::= \mathsf{O}\, X...y...\alpha...$$

Types A, kinds k, disciplines s, and (co)data declarations $decl$

$$A, B, C ::= X \mid \mathsf{F} \mid \lambda X{:}k.A \mid A\ B \qquad k, l ::= s \mid k \to l \qquad r, s, t ::= + \mid - \mid \sharp \mid \flat$$
$$decl ::= \mathbf{data}\, \mathsf{F}(X{:}k)... : s\, \mathbf{where}\, \mathsf{K} : (A{:}t... \vdash^{Y:s'...} \mathsf{F}\, X... \mid B{:}r...)...$$
$$\mid \mathbf{codata}\, \mathsf{G}(X{:}k)... : s\, \mathbf{where}\, \mathsf{O} : (A{:}t... \mid \mathsf{G}\, X... \vdash^{Y:s'...} B{:}r...)...$$

Figure 20: Syntax of System $\mathcal{CD}$: generalizing System $\mathcal{D}$ with fully dual (co)data.

constructors ($\mathsf{K}_1 \ldots \mathsf{K}_n$) for building values of F types. The way to read the signature of each constructor is that "inputs" are given on the left of the turnstyle ($\vdash$) and outputs are given on the right of the turnstyle. The main way this data declaration differs from a functional ADT is the availability of several different output types besides the main return type of the constructor. For example, the first constructor K_1 is parameterized by several terms of types $A_1 : t_1...$ (found on the left) and several coterms of types $B_1 : r_1...$ (found on the right), respectively, and will return a construction of type $\mathsf{F}\, X...$ (the main output on the right between $\vdash$ and $|$). The constructor K_1 is also parameterized by several types $Y_1...$ of kind $s_1...$. These types may be referred to in the types of the term and coterm arguments, but are hidden (*i.e.,* existentially quantified) to the consumer of the construction.

Codata type declarations are exactly symmetric to data type declarations, and have a very similar form:

$$\begin{aligned}&\mathbf{codata}\, \mathsf{G}(X{:}k)... : s\, \mathbf{where}\\ &\quad \mathsf{O}_1 : (A_1 : t_1... \mid \mathsf{G}\, X... \vdash^{Y_1:s_1...} B_1 : r_1...)\\ &\quad \ldots\\ &\quad \mathsf{O}_n : (A_n : t_n... \mid \mathsf{G}\, X... \vdash^{Y_n:s_n...} B_n : r_n...)\end{aligned}$$

As before, this declaration says that G is parameterized by several types and returns a type of the discipline s. But now, there are n different destructors ($\mathsf{O}_1 \ldots \mathsf{O}_n$) for building observations that use values of G types. The flow of information is still inputs on the left and outputs on the right, as before. For example, the first destructor O_1 is parameterized by several terms of types $A_1 : t_1...$ (found on the left) and several coterms of types $B_1 : r_1...$ (found on the right), respectively, and will consume a value of type $\mathsf{G}\, X...$ (the main input on the left between $|$ and $\vdash$). The type parameters $Y_1...$ to the observer O_1 are still brought into scope for (co)term parameters of O_1, but are now hidden (*i.e.,* universally quantified) to the value being consumed.

Several examples of data and codata type declarations are given in Figure 21. More specifically, these declarations give definitions for each of the connectives in the $\mathcal{D}$ core calculus from Section 7, which illustrates that the $\mathcal{CD}$ calculus completely subsumes the core calculus. As another example of an exotic declaration, we can declare function types as codata, as well the dual to the function type (known as subtraction or difference, and often

Simple (co)data types

$$\begin{array}{ll}
\mathbf{data}\,(X{:}{+}) \oplus (Y{:}{+}) : {+}\,\mathbf{where} & \mathbf{codata}\,(X{:}{-}) \,\&\, (Y{:}{-}) : {-}\,\mathbf{where} \\
\quad \iota_1 : (X{:}{+} \vdash X \oplus Y \mid\,) & \quad \pi_1 : (\,\mid X \,\&\, Y \vdash X{:}{-}) \\
\quad \iota_2 : (Y{:}{+} \vdash X \oplus Y \mid\,) & \quad \pi_2 : (\,\mid X \,\&\, Y \vdash Y{:}{-}) \\
\mathbf{data}\,0 : {+}\,\mathbf{where} & \mathbf{codata}\,\top : {-}\,\mathbf{where} \\
\mathbf{data}\,(X{:}{+}) \otimes (Y{:}{+}) : {+}\,\mathbf{where} & \mathbf{codata}\,(X{:}{-}) \parr (Y{:}{-}) : {-}\,\mathbf{where} \\
\quad (_,_) : (X{:}{+}, Y{:}{+} \vdash X \otimes Y \mid\,) & \quad [_,_] : (\,\mid X \parr Y \vdash X{:}{-}, Y{:}{-}) \\
\mathbf{data}\,1 : {+}\,\mathbf{where}\,() : (\vdash 1 \mid\,) & \mathbf{codata}\,\bot : {+}\,\mathbf{where}\,[] : (\,\mid \bot \vdash\,) \\
\mathbf{data} \ominus (X{:}{-}) : {+}\,\mathbf{where} & \mathbf{codata}\,\neg(X{:}{+}) : {-}\,\mathbf{where} \\
\quad \ominus : (\vdash \ominus X \mid X{:}{-}) & \quad \neg : (X{:}{+} \mid \neg X \vdash\,)
\end{array}$$

Quantifier (co)data types

$$\begin{array}{ll}
\mathbf{data}\,\exists_s(X{:}s{\to}{+}) : {+}\,\mathbf{where} & \mathbf{codata}\,\forall_s(X{:}s{\to}{-}) : {-}\,\mathbf{where} \\
\quad _\,\mathbin{⦶}\,_ : (X\ Y{:}{+} \vdash^{Y:s} \exists_s X) & \quad _\,@\,_ : (\,\mid \forall_s X \vdash^{Y:s} X\ Y{:}{-})
\end{array}$$

Polarity shift (co)data types

$$\begin{array}{ll}
\mathbf{data} \downarrow_s (X{:}s) : {+}\,\mathbf{where} & \mathbf{data}\ {}_s{\Uparrow}(X{:}{+}) : s\,\mathbf{where} \\
\quad \downarrow_s : (X{:}s \vdash \downarrow_s X) & \quad {}_s{\Uparrow} : (X{:}{+} \vdash\ {}_s{\Uparrow} X) \\
\mathbf{codata} \uparrow_s (X{:}s) : {-}\,\mathbf{where} & \mathbf{codata}\ {}_s{\Downarrow}\,\mathbf{where} \\
\quad \uparrow_s : (\,\mid \uparrow_s X \vdash X{:}s) & \quad {}_s{\Downarrow} : (\,\mid {}_s{\Downarrow} X \vdash X{:}{-})
\end{array}$$

Figure 21: (Co)Data declarations of the dual System $\mathcal{D}$ connectives.

written as $A - B$):

$$\begin{array}{ll}
\mathbf{codata}\,(X{:}{+}) \to (Y{:}{-}) : {-}\,\mathbf{where} & \mathbf{data}\,(X{:}{+}) \oslash (Y{:}{-}) : {+}\,\mathbf{where} \\
\quad _\cdot_ : (X{:}{+} \mid X \to Y \vdash Y{:}{-}) & \quad _ \oslash _ : (X{:}{+} \vdash X \oslash Y \mid Y{:}{-})
\end{array}$$

8.2. **Type system and semantics.** In order to finish our extension to $\mathcal{D}$, we also need to extends its type system and semantics. The type system for $\mathcal{CD}$ keeps the rules from Figures 13 and 14, and generalizes the form of each judgment to be parameterized by a set of declarations (denoted by $\mathcal{G}$) that replaces the $\mathcal{D}$ annotation. There are also connective-specific typing rules for each declaration in $\mathcal{G}$, given in Figure 22. Again, the special instances of these inference rule templates for the declarations in Figure 21 give exactly the right and left rules in Figures 15 to 17; fully dual (co)data declarations subsume the type system for the core $\mathcal{D}$ calculus.

Notice that the equational theory and operational semantics in Figures 18 and 19 automatically extend to accommodate user-defined data and codata types just by considering the extended notions of patterns p and copatterns q as given in Figure 20. Furthermore, the generalizations to weak-head normal terms and forcing contexts can be stated in terms of these extended (co)patterns as follows:

$$W ::= x \mid p[\rho] \mid \lambda\{q.c \mid ...\} \qquad\qquad F ::= \alpha \mid q[\rho] \mid \lambda\{p.c \mid ...\}$$

Given that $\mathcal{G}$ contains the data declaration

$$\mathbf{data}\,\mathsf{F}(X{:}k)... : s\,\mathbf{where}\,\mathsf{K}_i : (A_{ij} : t_{ij}\,\vec{j}. \vdash^{Y_{ij}:s'_{ij}\,\vec{j}.}\ \mathsf{F}\,X... \mid B_{ij} : r_{ij}\,\vec{j}.)\,\vec{i}.$$

we have the rules:

$$\frac{(\Theta \vdash_{\mathcal{G}} C'_j : s'_{ij})\,\vec{j}. \quad (\Gamma \mid e_j : B_{ij}[\rho] \vdash^{\Theta}_{\mathcal{G}}\ \Delta)\,\vec{j}. \quad (\Gamma \vdash^{\Theta}_{\mathcal{G}}\ v_j : A_{ij}[\rho] \mid \Delta)\,\vec{j}. \quad \rho = C/X..., C'_j/Y_{ij}\,\vec{j}.}{\Gamma \vdash^{\Theta}_{\mathcal{G}}\ \mathsf{K}_i\,C'_j\,\vec{j}.\ v_j\,\vec{j}.\ e_j\,\vec{j}. : \mathsf{F}\,C... \mid \Delta}\ \mathsf{F}R_i$$

$$\frac{c_i : (\Gamma, x_{ij} : A_{ij}[\rho]\,\vec{j}. \vdash^{\Theta, Y_{ij}:s'_{ij}\,\vec{j}.}_{\mathcal{G}}\ \alpha_{ij} : B_{ij}[\rho]\,\vec{j}., \Delta)\,\vec{i}. \quad \rho = C/X...}{\Gamma \mid \lambda\{(\mathsf{K}_i\,Y_{ij}\,\vec{j}.\alpha_{ij}\,\vec{j}.x_{ij}\,\vec{j}.).c_i\,\vec{i}.\} : \mathsf{F}\,C... \vdash^{\Theta}_{\mathcal{G}}\ \Delta}\ \mathsf{F}L$$

Given that $\mathcal{G}$ contains the codata declaration

$$\mathbf{codata}\,\mathsf{G}(X{:}k)... : s\,\mathbf{where}\,\mathsf{O}_i : (A_{ij} : t_{ij}\,\vec{j}. \mid \mathsf{G}\,X... \vdash^{Y_{ij}:s'_{ij}\,\vec{j}.}\ B_i : r_i)\,\vec{i}.$$

we have the rules:

$$\frac{(\Theta \vdash_{\mathcal{G}} C'_j : s'_{ij})\,\vec{j}. \quad (\Gamma \vdash^{\Theta}_{\mathcal{G}}\ v_j : A_{ij}[\rho] \mid \Delta)\,\vec{j}. \quad (\Gamma \mid e_j : B_{ij}[\rho] \vdash^{\Theta}_{\mathcal{G}}\ \Delta)\,\vec{j}. \quad \rho = C/X..., C'_j/Y_{ij}\,\vec{j}.}{\Gamma \mid \mathsf{O}_i\,C'_j\,\vec{j}.\ v_j\,\vec{j}.\ e_j\,\vec{j}. : \mathsf{G}\,C... \vdash^{\Theta}_{\mathcal{G}}\ \Delta}\ \mathsf{G}L_i$$

$$\frac{c_i : (\Gamma, x_{ij} : A_{ij}[\rho]\,\vec{j}. \vdash^{\Theta, Y_{ij}:s'_{ij}\,\vec{j}.}_{\mathcal{G}}\ \alpha_{ij} : B_{ij}[\rho]\,\vec{j}., \Delta)\,\vec{i}. \quad \rho = C/X...}{\Gamma \vdash^{\Theta}_{\mathcal{G}}\ \lambda\{[\mathsf{O}_i\,Y_{ij}\,\vec{j}.x_{ij}\,\vec{j}.\alpha_{ij}\,\vec{j}.].c_i\,\vec{i}.\} : \mathsf{F}\,C... \mid \Delta}\ \mathsf{G}R$$

Figure 22: System $\mathcal{CD}$ typing rules for user-defined classical data and codata types.

The only aspect of the semantics that is left to be stated is how to handle the occurrence of non-value or non-covalue parameters to constructors and destructors. We will handle this possibility in the same way as we originally did in Section 2: define a translation into an appropriate sub-syntax which rules out this possibility. We can run arbitrary commands by first translating them and then applying the operational semantics to the well-behaved sub-syntax. And to reason about equality, we will identify all commands, terms, and coterms with their translation in the equational theory. But this translation will serve another purpose besides just defining the semantics of $\mathcal{CD}$: it is the first step of compiling all of $\mathcal{CD}$ into the core $\mathcal{D}$.

9. Compiling Classical (Co)Data Types Into the Core $\mathcal{D}$

Our goal now is to demonstrate that all the new features of the extensible $\mathcal{CD}$ calculus can be expressed within the core $\mathcal{D}$ calculus. We will do this by showing how to compile all of $\mathcal{CD}$ into $\mathcal{D}$ using only macro-expansions, which shows that the two languages are equally expressive [Fel91]. This compilation process involves two phases: (1) a translation into a focused sub-syntax defined as a generalization of focusing described in Sections 2, 3 and 5, and (2) a translation of user-defined data and codata types into the core connectives in $\mathcal{D}$. The composition of these two steps gives an embedding of $\mathcal{CD}$ inside $\mathcal{D}$.

9.1. Focusing the syntax.

In order to identify the focused sub-syntax of $\mathcal{CD}$, we have to ensure that constructors (K) and destructors (O) are only ever applied to values and covalues. The key idea is that the correct notion of value and covalue for each parameter can be found from the signature of every constructor and destructor. That means there is a different sub-syntax for each set of declarations, $\mathcal{G}$.

$$[\![\mathsf{F}]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq \lambda X{:}k{....}_s\Uparrow((\exists Y_{ij}{:}s'_{ij}.^j_{..}.((\ominus(\uparrow_{r_{ij}} B_{ij})) \otimes .^j_{..}((\downarrow_{t_{ij}} A_{ij}) \otimes .^j_{..}1))) \oplus .^i_{..}0)$$
$$[\![\mathsf{G}]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq \lambda X{:}k{....}_s\Downarrow((\forall Y_{ij}{:}s'_{ij}.^j_{..}.((\neg(\downarrow_{t_{ij}} A_{ij})) \parr .^j_{..}((\uparrow_{r_{ij}} B_{ij}) \parr .^j_{..}\bot))) \mathbin{\&} .^i_{..}\top)$$

$$[\![\mathsf{K}_i\, Y ...\ \alpha ...\ x ...]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq {}_s\Uparrow\left(\iota^i_2\left(\iota_1\left(Y \mathbin{⓪} ... \left(\ominus[\uparrow_r \alpha], ... \left(\downarrow_t x, ...()\right)\right)\right)\right)\right)$$
$$[\![\mathsf{O}_i\, Y ...\ x ...\ \alpha ...]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq {}_s\Downarrow\left[\pi^i_2\left[\pi_1\left[Y \mathbin{@} ... \left[\neg[\downarrow_t x], ... \left[\uparrow_r \alpha, ...[]\right]\right]\right]\right]\right]$$
$$[\![\lambda\{q_i.c_i.^i_{..}\}]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq \lambda\{[\![q_i]\!]^{\mathcal{D}}_{\mathcal{G}}.[\![c_i]\!]^{\mathcal{D}}_{\mathcal{G}}.^i_{..}\} \qquad [\![p[\rho]]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq [\![p]\!]^{\mathcal{D}}_{\mathcal{G}}[[\![\rho]\!]^{\mathcal{D}}_{\mathcal{G}}]$$
$$[\![\mathcal{K}\{p_i.c_i.^i_{..}\}]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq \mathcal{K}\{[\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}.[\![c_i]\!]^{\mathcal{D}}_{\mathcal{G}}.^i_{..}\} \qquad [\![q[\rho]]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq [\![q]\!]^{\mathcal{D}}_{\mathcal{G}}[[\![\rho]\!]^{\mathcal{D}}_{\mathcal{G}}]$$
$$[\![C/X{:}k...V_t/x{::}t...E_r/\alpha{::}r...]\!]^{\mathcal{D}}_{\mathcal{G}} \triangleq [\![C]\!]^{\mathcal{D}}_{\mathcal{G}}/X{:}k...[\![V_t]\!]^{\mathcal{D}}_{\mathcal{G}}/x{::}t...[\![E_r]\!]^{\mathcal{D}}_{\mathcal{G}}/\alpha{::}r...$$

$$\iota^0_i(p) = p \qquad \iota^{n+1}_i(p) = \iota_i(\iota^n_i(p)) \qquad \pi^0_i[q] = q \qquad \pi^{n+1}_i[q] = \pi_i[\pi^n_i[q]]$$

Figure 23: Compiling System $\mathcal{CD}$ into System $\mathcal{D}$.

Definition 9.1 (Focused sub-syntax)**.** The *$\mathcal{G}$-focused sub-syntax* of the $\mathcal{CD}$ calculus restricts constructions to the form $\mathsf{K}\,C...E_r...V_t...$ and destructions to the form $\mathsf{O}\,C...E_r...V_t...$ when, respectively,

$$\mathsf{K} : \left(A : t... \vdash^{Y:s,...} \mathsf{F}\ X... \mid B : r...\right) \in \mathcal{G} \qquad \mathsf{O} : \left(A : t... \mid \mathsf{G}\ X... \vdash^{Y:s,.....} B : r...\right) \in \mathcal{G}$$

We can then compile any $\mathcal{CD}$ expression into a corresponding $\mathcal{G}$-focused expression through the following macro-expansion, that just names the parameters to every constructor and destructor:

$$(\mathsf{K}\,C...e...v...)^{\varsigma} = \mu\alpha.\left\langle \mu\beta.\ldots.\left\langle v^{\varsigma} \middle\| \tilde{\mu}y.\ldots.\langle \mathsf{K}\,C...\beta...y...\|\alpha\rangle \right\rangle \middle\| e^{\varsigma} \right\rangle$$
$$(\mathsf{O}\,C...v...e...)^{\varsigma} = \tilde{\mu}x.\left\langle v^{\varsigma} \middle\| \tilde{\mu}y.\ldots.\left\langle \mu\beta.\ldots.\langle x \| \mathsf{O}\,C...y...\beta...\rangle \right\rangle \middle\| e^{\varsigma} \right\rangle$$

where the type and discipline annotations on each command can be derived from the signature of K and O.

Recall that the above focusing translation is part of the definition of the semantics for $\mathcal{CD}$, which says how to handle terms and coterms which do not fit within the focused sub-syntax. In the operational semantics, we can just compile a command c to c^{ς} prior to evaluation. And formally stated in the equational theory, we make the additional assertion that $c = c^{\varsigma} : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta)$ for any $c : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta)$ (and analogously for terms and coterms).

9.2. **Translating user-defined types.** We are now prepared to give a translation of arbitrary user-defined (co)data types into the core $\mathcal{D}$ connectives. The translation is parameterized by a global environment $\mathcal{G}$ so that we know the overall shape of each connective, and is given in Figure Figure 23. The translation of individual data and codata type constructors assume that $\mathcal{G}$ contains a data or codata type of the forms listed in Figure 22. The translation of data types wraps every component type in the appropriate shift ($\downarrow$ for inputs and $\uparrow$ for outputs) into the realm of positive types, $\ominus$-negates every additional output of each constructor, combines the components of each constructor with a nested $\otimes$ pair, combines the choice between different constructors with a nested $\oplus$ sum, and introduces all the private types with $\exists$ quantifiers. Finally, the translated data type is injected into the desired discipline with the outer-most $\Uparrow$ shift to come back from the realm of positive types. The translation of multi-output codata is exactly dual to the data

translation, making a round trip through the realm of negative types for the purpose of using the basic codata building blocks. We can generalize this translation to arbitrary types homomorphically in all other cases (*i.e.,* $[\![A\ B]\!]^{\mathcal{D}}_{\mathcal{G}} = [\![A]\!]^{\mathcal{D}}_{\mathcal{G}}\ [\![B]\!]^{\mathcal{D}}_{\mathcal{G}}$, *etc.*), iterating as needed to translate the connectives introduced by declarations. The translation of (co)patterns, (co)pattern-matching objects, and (co)data structures follows the translation of types.

For example, with declarations of function and subtraction types

$$\mathbf{codata}\,(X{:}+) \to (Y{:}-) : -\,\mathbf{where} \qquad\qquad \mathbf{data}\,(X{:}+) \oslash (Y{:}-) : +\,\mathbf{where}$$
$$\_ \cdot \_ : (X{:}+ \mid X \to Y \vdash Y{:}-) \qquad\qquad \_ \oslash \_ : (X{:}+ \vdash X \oslash Y \mid Y{:}-)$$

in the set of declarations $\mathcal{G}$, we get the following translations:

$$[\![\to]\!]^{\mathcal{D}}_{\mathcal{G}} = \lambda X{:}+.\lambda Y{:}-.\_ \Downarrow ((\neg(\downarrow_+ X) \parr (\uparrow_- Y \parr \bot)) \,\&\, \top)$$
$$[\![\oslash]\!]^{\mathcal{D}}_{\mathcal{G}} = \lambda X{:}+.\lambda Y{:}-.\ _+ \Uparrow ((\ominus(\uparrow_- Y) \otimes (\downarrow_+ X \otimes 1)) \oplus 0)$$

But these are rather heavy translation for such simple connectives. Instead, we can sometimes simplify the results of translation based on the idea of *type isomorphisms*, which we will explore next in Section 10. For these specific types, we have the following isomorphic translations:

$$[\![A \to B]\!]^{\mathcal{D}}_{\mathcal{G}} \approx (\neg[\![A]\!]^{\mathcal{D}}_{\mathcal{G}}) \parr [\![B]\!]^{\mathcal{D}}_{\mathcal{G}} \qquad\qquad [\![A \oslash B]\!]^{\mathcal{D}} \approx (\ominus[\![B]\!]^{\mathcal{D}}_{\mathcal{G}}) \otimes [\![A]\!]^{\mathcal{D}}_{\mathcal{G}}$$

9.3. **Correctness of compilation.** The only thing remaining is to ensure that this compilation of values and covalues preserves the semantics of the original program that used custom data and codata types: the observable result of running a program (that is, the set of needed variables and covariables) is exactly the same before or after compilation, and all the equalities of $\mathcal{CD}$ are preserved in $\mathcal{D}$.

Theorem 9.2 (Operational Correspondence)**.** *For non-cyclic $\mathcal{G}$, command c in $\mathcal{CD}$, and non-empty set of (co)variables R, the following statements are equivalent:*

- *$c \mapsto\!\!\!\!\twoheadrightarrow c'$ for some finished c' in $\mathcal{CD}$ such that $\mathrm{NV}(c') = R$, and*
- *$[\![c]\!]^{\mathcal{D}}_{\mathcal{G}} \mapsto\!\!\!\!\twoheadrightarrow c'$ for some finished c' in $\mathcal{D}$ such that $\mathrm{NV}(c') = R$.*

Proof. We apply techniques used previously in [DMAV14] and [DA14a], which allow us to infer one direction of the bi-implication from the other. First, observe the following properties of the compilation translation.

a) *Heaps*: For all c and H, there is a heap context H' such that $[\![H[c]]\!]^{\mathcal{D}}_{\mathcal{G}} =_\alpha H'[[\![c]\!]^{\mathcal{D}}_{\mathcal{G}}]$.
b) *(Co)Values*: For all V_s and E_s, $[\![V_s]\!]^{\mathcal{D}}_{\mathcal{G}}$ is an s-value and $[\![E_s]\!]^{\mathcal{D}}_{\mathcal{G}}$ is an s-covalue.
c) *Need*: $\mathrm{NV}(c) = \mathrm{NV}([\![c]\!]^{\mathcal{D}}_{\mathcal{G}})$.
d) *Substitution*: For all commands c and substitutions ρ, $[\![c[\rho]]\!]^{\mathcal{D}}_{\mathcal{G}} =_\alpha [\![c]\!]^{\mathcal{D}}_{\mathcal{G}}[[\![\rho]\!]^{\mathcal{D}}_{\mathcal{G}}]$.
e) *Reduction*: For all c and c', if $c \mapsto c'$ then $[\![c]\!]^{\mathcal{D}}_{\mathcal{G}} \mapsto\!\!\!\!\twoheadrightarrow [\![c']\!]^{\mathcal{D}}_{\mathcal{G}}$ in *one or more* steps.

Part (a) follows from the facts that the translation is compositional and does not change the signs on commands. Furthermore, parts (b), (c), and (d) follow from part (a) and the facts that the translation is both compositional and hygienic [DA14a] (*i.e.,* it does not capture or escape free (co)variables of sub-expressions). Finally, part (e) follows from part (a) and the fact that each operational step is preserved by the translation. We know that the rules

for $\beta_\mu^{\pm\sharp}$, $\beta_{\tilde{\mu}}^{\pm\flat}$, $\phi_\mu^\flat$, and $\phi_{\tilde{\mu}}^\sharp$ are preserved by translation due to parts (b) and (d). That leaves only the β_p and β_q rules, which take the following form after translation:

$$(\llbracket\beta_q\rrbracket_{\mathcal{G}}^{\mathcal{D}}) \qquad \left\langle \lambda\{\cdots \mid \llbracket q\rrbracket_{\mathcal{G}}^{\mathcal{D}}.c \mid \ldots\} \middle| s \middle| \llbracket q\rrbracket_{\mathcal{G}}^{\mathcal{D}}[\rho] \right\rangle \mapsto\!\!\!\!\rightarrow c[\rho]$$

$$(\llbracket\beta_p\rrbracket_{\mathcal{G}}^{\mathcal{D}}) \qquad \left\langle \llbracket p\rrbracket_{\mathcal{G}}^{\mathcal{D}}[\rho] \middle| s \middle| \curlywedge\{\cdots \mid \llbracket p\rrbracket_{\mathcal{G}}^{\mathcal{D}}.c \mid \ldots\} \right\rangle \mapsto\!\!\!\!\rightarrow c[\rho]$$

Both of these reductions can be verified through routine calculation of the smaller-step pattern-matching rules from Figure 19 by expanding out the nested (co)patterns from the translation (see Appendix A for more details), which confirms that they always take *at least* one step. It then follows that if $c \mapsto\!\!\!\!\rightarrow c'$ and c' is finished in $\mathcal{CD}$ with $\mathrm{NV}(c') = R$, we know by transitivity of part (e) that $\llbracket c\rrbracket_{\mathcal{G}}^{\mathcal{D}} \mapsto\!\!\!\!\rightarrow \llbracket c'\rrbracket_{\mathcal{G}}^{\mathcal{D}}$ and by part (c) $\llbracket c'\rrbracket_{\mathcal{G}}^{\mathcal{D}}$ is finished in $\mathcal{D}$ with $\mathrm{NV}(\llbracket c'\rrbracket_{\mathcal{G}}^{\mathcal{D}}) = \mathrm{NV}(c')$.

The reverse direction must also hold because the operational semantics of $\mathcal{D}$ and $\mathcal{CD}$ are deterministic [DMAV14], which implies two more facts about the uniqueness of their results:

(1) There is an infinite reduction sequence starting with c if and only if there is no finished or stuck c' such that $c \mapsto\!\!\!\!\rightarrow c'$.
(2) For every command c, there is *at most one* finished or stuck c' such that $c \mapsto\!\!\!\!\rightarrow c'$.

Suppose that $\llbracket c\rrbracket_{\mathcal{G}}^{\mathcal{D}} \mapsto\!\!\!\!\rightarrow c'$ and c' is finished in $\mathcal{D}$ with $\mathrm{NV}(c') = R$. It cannot be that there is an infinite reduction sequence beginning from c, because then part (e) would imply there is an infinite reduction from $\llbracket c\rrbracket_{\mathcal{G}}^{\mathcal{D}}$, contradicting the first uniqueness fact. So there must be some finished or stuck c'' in $\mathcal{CD}$ such that $c \mapsto\!\!\!\!\rightarrow c''$. By part (e) we know $\llbracket c\rrbracket_{\mathcal{G}}^{\mathcal{D}} \mapsto\!\!\!\!\rightarrow \llbracket c''\rrbracket_{\mathcal{G}}^{\mathcal{D}}$ in $\mathcal{CD}$. And from the second uniqueness fact and part (c), it must be that $\llbracket c''\rrbracket_{\mathcal{G}}^{\mathcal{D}}$ is c' and $\mathrm{NV}(c'') = \mathrm{NV}(\llbracket c''\rrbracket_{\mathcal{G}}^{\mathcal{D}}) = \mathrm{NV}(c') = R$. □

Theorem 9.3 (Equational Soundness)**.** *For non-cyclic $\mathcal{G}$, if $c = c' : (\Gamma \vdash_{\mathcal{G}}^{\Theta} \Delta)$ then $\llbracket c\rrbracket_{\mathcal{G}}^{\mathcal{D}} = \llbracket c'\rrbracket_{\mathcal{G}}^{\mathcal{D}} : (\llbracket\Gamma\rrbracket_{\mathcal{G}}^{\mathcal{D}} \vdash_{\mathcal{D}}^{\Theta} \llbracket\Delta\rrbracket_{\mathcal{G}}^{\mathcal{D}})$.*

Proof. Follows similarly to Theorem 9.2. The equational theory introduces the $\chi^\sharp$ and $\chi^\flat$ axioms, which are unaffected, and the generalized β_μ^s and $\beta_{\tilde{\mu}}^s$ axioms, which are still sound due to the fact that translation commutes over substitution and preserves the (co)values of every discipline. It also introduces the η_p and η_q axioms which express extensionality of (co)data abstractions, and take the following form after translation:

$$(\llbracket\eta_q\rrbracket_{\mathcal{G}}^{\mathcal{D}}) \qquad \lambda\left\{ \llbracket q\rrbracket_{\mathcal{G}}^{\mathcal{D}}.\langle x|s|\llbracket q\rrbracket_{\mathcal{G}}^{\mathcal{D}}\rangle \ldots \right\} = x$$

$$(\llbracket\eta_p\rrbracket_{\mathcal{G}}^{\mathcal{D}}) \qquad \curlywedge\left\{ \llbracket p\rrbracket_{\mathcal{G}}^{\mathcal{D}}.\langle \llbracket p\rrbracket_{\mathcal{G}}^{\mathcal{D}}|s|\alpha\rangle \ldots \right\} = \alpha$$

As before, both of these equations can be verified from the smaller-step rules from Figure 18 by expanding out the nested (co)patterns from the translation (see Appendix A). Finally, the inference rules of the equational theory (reflexivity, transitivity, symmetry, and compatibility) are sound due to the fact that the translation is compositional [DA14a]. □

10. Correct Encodings as Type Isomorphisms

The notion of compilation correctness given above is good for a whole-program transformation: the semantics of a $\mathcal{CD}$ program (represented as a command) is preserved by compilation inside of a $\mathcal{D}$ program. However, what if we instead want to apply the translation of (co)data types locally? Suppose we translate, not as a way to compile an entire program, but as a way to encoding only to a particular sub-(co)term or definition in a larger program. Correctness of a local encoding requires more than just correctness of a global compilation. A global compiler can control both the implementations ((co)terms) and clients (contexts) to respect some invariants that are essential for correctness. But a local encoding can only control the implementation; it has no control over its clients. Therefore, to be correct, an encoding must make sure that it gives an implementation with the exact same semantics as the source for *every* possible client context that might be written in the resulting code, not just the clients that happen to be aware of the details of the encoding.

So what is the appropriate notion of correctness for an encoding that translates fully-dual user-defined data and codata types into the core $\mathcal{D}$ connectives? We want to be sure that every equality between the program that was originally written is preserved by the encoding; otherwise, optimizations that were valid on the original program may be broken by an unfortunate encoding. This kind of idea is captured by an *equational correspondence* [SF93] between the source and target calculi of the encoding process, which states that two expressions are equal exactly when they are equal after translation.

However, our situation is a little more delicate: by encoding arbitrarily many types into a small set of core connectives in $\mathcal{D}$, we might accidentally equate two different types in the source language, which in turn would accidentally equate their terms and coterms! So we need to use more finesse when reasoning about the relationship between the source and target languages which takes the different types into account. To that end, we will apply the idea of a *type isomorphism*—an invertible mapping between two types—which will let us relate the original declared type with its encoding. This gives us a principled method of framing encoding correctness, even when multiple types are collapsed into one.

The only stumbling block with using type isomorphisms to establish correctness is our use of type abstractions: the $\forall$ and $\exists$ quantifications over types makes it difficult to compose type isomorphisms together into a big step encoding. However, the quantifiers of $\mathcal{D}$ have some extra structure: they are *parametric*, meaning that processes with abstract types must do the same thing for every specialization. Therefore, we generalize the usual notion of type isomorphism to take this parametricity into account, so that the witnesses to isomorphisms are likewise parametric, with a *single* witness that applies uniformly to any particular instantiation. As a result, we can compose parametric type isomorphisms freely even under quantifiers, which lets us incrementally build up the correctness of complex encodings into the core $\mathcal{D}$ calculus.

10.1. Parametric type isomorphism.

The encodings we gave make a number of choices: the orders between different constructors or destructors with respect to one another, between the components of constructors and destructors, *etc.* Did we make the correct choice to preserve the semantics of the original type? Or does this difference even matter? It turns out any of these choices are correct: each would give types that are *isomorphic* to one another.

Definition 10.1 (Parametric Type Isomorphism)**.** A *parametric isomorphism* between two types $\Theta \vdash_{\mathcal{G}} A : k$ and $\Theta \vdash_{\mathcal{G}} B : k$, written $\Theta \vDash_{\mathcal{G}} A \approx B : k$, is defined by induction on k:

Positive algebraic laws (for $A, B, C : {}_+$):

$$(A \oplus B) \oplus C \approx A \oplus (B \oplus C) \qquad 0 \oplus A \approx A \qquad A \oplus B \approx B \oplus A$$
$$(A \otimes B) \otimes C \approx A \otimes (B \otimes C) \qquad 1 \otimes A \approx A \qquad A \otimes B \approx B \otimes A$$
$$A \otimes (B \oplus C) \approx (A \otimes B) \oplus (A \otimes C) \qquad A \otimes 0 \approx 0$$

Negative algebraic laws (for $A, B, C : {}_-$):

$$(A \,\&\, B) \,\&\, C \approx A \,\&\, (B \,\&\, C) \qquad \top \,\&\, A \approx A \qquad A \,\&\, B \approx B \,\&\, A$$
$$(A \parr B) \parr C \approx A \parr (B \parr C) \qquad \bot \parr A \approx A \qquad A \parr B \approx B \parr A$$
$$A \parr (B \,\&\, C) \approx (A \parr B) \,\&\, (A \parr C) \qquad A \parr \top \approx \top$$

De Morgan laws (with $A, B : {}_+$ and $C, D : {}_-$):

$$\neg(A \oplus B) \approx (\neg A) \,\&\, (\neg B) \quad \neg(A \otimes B) \approx (\neg A) \parr (\neg B) \quad \neg 0 \approx \top \quad \neg 1 \approx \bot \quad \neg(\ominus C) \approx C$$
$$\ominus(C \,\&\, D) \approx (\ominus C) \oplus (\ominus D) \quad \ominus(C \parr D) \approx (\ominus C) \otimes (\ominus D) \quad \ominus\top \approx 0 \quad \ominus\bot \approx 1 \quad \ominus(\neg A) \approx A$$

Shift laws (for $A : {}_+$ and $B : {}_-$):

$$\downarrow_- B \approx {}_+\Downarrow B \qquad \uparrow_+ A \approx {}_-\Uparrow A \qquad \downarrow_+ A \approx A \approx {}_+\Uparrow A \qquad \uparrow_- B \approx B \approx {}_-\Downarrow B$$

Quantifier laws (for $A, B : {}_+$ and $C, D : {}_-$ where $X \notin FV(B) \cup FV(D)$):

$$\forall X{:}s.\forall Y{:}t.C \approx \forall Y{:}t.\forall X{:}s.C \qquad \exists X{:}s.\exists Y{:}t.A \approx \exists Y{:}t.\exists X{:}s.A$$
$$\forall X{:}s.D \approx D \qquad \exists X{:}s.B \approx B$$
$$(\forall X{:}s.C) \parr D \approx \forall X{:}s.(C \parr D) \qquad (\exists X{:}s.A) \otimes B \approx \exists X{:}s.(A \otimes B)$$
$$\ominus(\forall X{:}s.C) \approx \exists X{:}s.(\ominus C) \qquad \neg(\exists X{:}s.A) \approx \forall X{:}s.(\neg A)$$

Figure 24: Core System $\mathcal{D}$ type isomorphism laws.

- $\Theta \vDash_{\mathcal{G}} A \approx B : s$ when there are commands $c' : (x : A \vdash^{\Theta}_{\mathcal{G}} \beta : B)$ and $c : (y : B \vdash^{\Theta}_{\mathcal{G}} \alpha : A)$ such that the following equalities are derivable:
$$\left\langle \mu\beta.c' \middle| B{:}s \middle| \tilde{\mu}y.c \right\rangle = \langle x | A{:}s | \alpha \rangle : (x : A \vdash^{\Theta}_{\mathcal{G}} \alpha : A)$$
$$\left\langle \mu\alpha.c \middle| A{:}s \middle| \tilde{\mu}x.c' \right\rangle = \langle y | B{:}s | \beta \rangle : (y : B \vdash^{\Theta}_{\mathcal{G}} \beta : B)$$
- $\Theta \vDash_{\mathcal{G}} A \approx B : k \to l$ when $\Theta, X : k \vDash_{\mathcal{G}} A\ X \approx B\ X : l$ (for any $X \notin \Theta$)

This notion of parametric type isomorphism lets us formally state how many representations of types are actually equivalent to one another. For example, we have the various type isomorphism laws in Figure 24. Some of these laws take a familiar form from type theory—like the algebraic laws—but they are stronger in $\mathcal{D}$ than in a conventional programming language: side effects commonly break these isomorphisms, but here they hold even in the presence of effects. The De Morgan laws are standard from classical logic, but they too are much stronger properties in this setting: they assert not just equi-provability that only guarantees well-typed mappings, but also assert that those mappings are value-preserving inverses. This is especially different from intuitionistic logic and pure functional languages, which do not have all the De Morgan laws in either sense. Intuitively, the problem is intuitionistic logics and languages only have the $\neg$-form of negation (which is sometimes defined as the isomorphic $\neg A \approx A \to \bot$), but not $\ominus$, which is why negation is not involutive (*i.e.,* $\neg(\downarrow(\neg A)) \not\approx A$) and does not distribute over products (*i.e.,* $\neg(\downarrow(C \,\&\, D)) \not\approx (\downarrow\neg C) \oplus (\downarrow\neg C)$).

But the fully dual, classical setting is expressive enough to capture these properties using both forms of involutive negation.

Type isomorphisms also let us state how some of the shift connectives are redundant, as shown by the shift laws. In particular, within the positive ($+$) and negative ($-$) subset, there are only two shifts of interest since the two different shifts between $-$ and $+$ are isomorphic, and the identity shifts on $+$ and $-$ are isomorphic to an identity on types. But clearly the shifts involving $\sharp$ are not isomorphic, since they are all different kinds to one another and the identity function on $\sharp$ types. Recognizing that sometimes the generic encoding uses identity shifts which can be elided up to isomorphism, the generic encodings of $[\![\to]\!]^{\mathcal{D}}_{\mathcal{G}}$ and $[\![\oslash]\!]^{\mathcal{D}}_{\mathcal{G}}$ are isomorphic to the simplified ones given earlier.

Finally, due to the parametric nature of type isomorphisms in Definition 10.1, we can now state some laws of the quantifiers, namely that adjacent pairs of the same quantifier can be reordered, unused quantifiers can be deleted, and we can distribute some other connectives over quantifiers. The most interesting of these is the relationship between quantifiers and negation. Intuitionistic languages have an isomorphism corresponding to $\neg(\exists X{:}s.A) \approx \forall X{:}s.(\neg A)$ but reject the reverse $\neg(\forall X{:}s.C) \approx \exists X{:}s.(\neg C)$. That's because every canonical proof of an existential in intuitionistic logic must present a specific witness: pointing out the witness to a proof is stronger than saying it is impossible that one cannot exist. Thus, the constructive notion of existentials corresponds to the invertible mapping between existential parameters of the form $\neg(B \mathbin{⅋} W)$ (isomorphic to the more familiar function call $f\ (B \mathbin{⅋} V)$) and $B \mathbin{@} \neg W$ (isomorphic to the System F application $(f\ B)\ V$).

The polarized treatment of negation helps to illuminate why the non-intuitionistic isomorphism is objectionable: it is not even well-formed! The correct form of the second isomorphism is $\ominus(\forall X{:}s.C) \approx \exists X{:}s.(\ominus C)$, which uses the other negation not found in intuitionistic logic, and which is quite a different statement. More concretely, it says that a canonical proof of an $\exists$ is the same as a canonical *refutation* of a $\forall$. In other words, the refutation of a $\forall$ is a specific counter-example, which is the same as saying that there exists a witness to prove the negative. Exhibiting a real counter-example is much stronger than merely asserting the impossibility that no such counter-example exists. The equivalent strength of existential witnesses and universal counter-examples corresponds to an invertible mapping between captured specialization call stacks of the form $\ominus(B \mathbin{@} F)$ and existential packages containing a captured call stack $B \mathbin{⅋} (\ominus F)$.

10.2. **Parametric type isomorphism is an equivalence relation.** Type isomorphisms are easier to work with than global transformations because they are compositional: small isomorphisms can be composed together to get bigger ones. This is due to the fact that they are *equivalence relations*, meaning a reflexive, symmetric, and transitive relation between types of the same kind. However, this fact does not come for free, we must actually show that the composite of inverse mappings are still inverses. It turns out that in terms of computation, this idea is captured in exactly one place: the χ law from the equational theory (Figure 18) that reassociates variable and covariable bindings.

Theorem 10.2 (Equivalence relation)**.** *Parametric isomorphism is an equivalence relation, meaning*

a) Reflexivity*:* $\Theta \vDash_{\mathcal{G}} A \approx A : k$ *if* $\Theta \vdash_{\mathcal{G}} A : k$*,*

b) Symmetry*:* $\Theta \vDash_{\mathcal{G}} A \approx B : k$ *if and only if* $\Theta \vDash_{\mathcal{G}} B \approx A : k$*, and*

c) Transitivity*: if* $\Theta \vDash_{\mathcal{G}} A \approx B : k$ *and* $\Theta \vDash_{\mathcal{G}} B \approx C : k$ *then* $\Theta \vDash_{\mathcal{G}} A \approx C : k$*.*

Proof. First, consider special case when $k = s$. Symmetry is immediate by Definition 10.1. Reflexivity is witnessed by the identity command $\langle x|A{:}s|\alpha\rangle : (x : A \vdash_{\mathcal{G}} \alpha : A)$ in both directions, because

$$\left\langle \mu\alpha.\, \langle x|A{:}s|\alpha\rangle \middle| A{:}s \middle| \tilde{\mu}x.\, \langle x|s{:}|\alpha\rangle \right\rangle =_{\eta_\mu \eta_{\tilde{\mu}}} \langle x|A{:}s|\alpha\rangle$$

Transitivity is the most difficult property to prove, and requires the use of χ^s equality (which is an axiom for $s = \sharp$ and $\flat$, and can be derived from β_μ or $\beta_{\tilde{\mu}}$ for $s = +$ and $-$). Suppose that we have the following witnesses to the assumed isomorphisms, respectively:

$$c_B : (x : A \vdash^{\Theta}_{\mathcal{G}} \beta : B) \qquad c_A : (y : B \vdash^{\Theta}_{\mathcal{G}} \alpha : A)$$
$$c_C : (y' : B \vdash^{\Theta}_{\mathcal{G}} \gamma : C) \qquad c'_B : (z : C \vdash^{\Theta}_{\mathcal{G}} \beta' : B)$$

We can then use these to form the following witnesses to $\Theta \vDash_{\mathcal{G}} A \approx C : s$:

$$c'_C = \left\langle \mu\beta.c_B \middle| B{:}s \middle| \tilde{\mu}y'.c_C \right\rangle : (x : A \vdash^{\Theta}_{\mathcal{G}} \gamma : C) \quad c'_A = \left\langle \mu\beta'.c'_B \middle| B{:}s \middle| \tilde{\mu}y.c_A \right\rangle : (z : C \vdash^{\Theta}_{\mathcal{G}} \alpha : A)$$

Both compositions of the above are the identity command (with only one direction shown as the other is dual) like so:

$$\begin{aligned}
\left\langle \mu\alpha.c'_A \middle| A{:}s \middle| \tilde{\mu}x.c'_C \right\rangle &= \left\langle \mu\alpha.\, \left\langle \mu\beta'.c'_B \middle| B{:}s \middle| \tilde{\mu}y.c_A \right\rangle \middle| A{:}s \middle| \tilde{\mu}x.\, \left\langle \mu\beta.c_B \middle| B{:}s \middle| \tilde{\mu}y'.c_C \right\rangle \right\rangle \\
&=_{\chi^s} \left\langle \mu\beta'.c'_B \middle| B{:}s \middle| \tilde{\mu}y.\, \left\langle \mu\alpha.c_A \middle| A{:}s \middle| \tilde{\mu}x.\, \left\langle \mu\beta.c_B \middle| B{:}s \middle| \tilde{\mu}y'.c_C \right\rangle \right\rangle \right\rangle \\
&=_{\chi^s} \left\langle \mu\beta'.c'_B \middle| B{:}s \middle| \tilde{\mu}y.\, \langle \mu\beta.\, \langle \mu\alpha.c_A | A{:}s | \tilde{\mu}x.c_B \rangle | B{:}s | \tilde{\mu}y'.c_C \right\rangle \right\rangle \\
&=_{iso} \left\langle \mu\beta'.c'_B \middle| B{:}s \middle| \tilde{\mu}y.\, \langle \mu\beta.\, \langle \beta | B{:}s | y \rangle | B{:}s | \tilde{\mu}y'.c_C \right\rangle \right\rangle \\
&=_{\eta_\mu \eta_{\tilde{\mu}}} \left\langle \mu\beta'.c'_B \middle| B{:}s \middle| \tilde{\mu}y'.c_C \right\rangle \\
&=_{iso} \langle z|C{:}s|\gamma\rangle
\end{aligned}$$

We can then finish by induction on a general k. Given that $k = l_1 \to \cdots \to l_n \to s$, then we can apply both sides of each type isomorphism to the fresh type variables $X_1 : l_1 \ldots X_n : l_n$, to get an equivalent isomorphism at kind s. □

10.3. **Parametric type isomorphism is a congruence relation.** But transitivity isn't the only form of compositionality; we also want the ability to apply type isomorphisms inside any context of a larger type. In other words, we want to be sure that isomorphisms are a *congruence relation* between types, meaning that it is compatible with all the type-forming operations. That way, we would know that for any context C that could surround both A and B, if $A \approx B$ then $C[A] \approx C[B]$.

There are several cases of compatibility to consider based on the grammar of types. The most fundamental case of compatibility is with the basic connectives that form types. For these, we can show individually that each of the finitely-many dual core $\mathcal{D}$ connectives are compatible with isomorphism; this is enough since every other connective can be encoded into these. This would ordinarily be difficult to show for the quantifiers ($\forall$ and $\exists$), because there might be different mappings to the isomorphisms for each specialization of the abstract type. However, the notion of parametric type isomorphism is stronger, and stipulates that the *same* mapping is used for every such specialization, which gives a *uniform* witness to isomorphisms found inside quantifiers.

Theorem 10.3 (Compatibility with $\mathcal{D}$ Connectives)**.** *Parametric type isomorphism is compatible with the core connectives of $\mathcal{D}$,* i.e., *the following isomorphisms hold for any $\mathcal{G}$ extending $\mathcal{D}$:*

- *For any $\Theta \vDash_{\mathcal{G}} A_1 \approx A_2 : +$ and $\Theta \vDash_{\mathcal{G}} B_1 \approx B_2 : +$ we have*

$$\Theta \vDash_{\mathcal{G}} A_1 \oplus B_1 \approx A_2 \oplus B_2 : + \qquad \Theta \vDash_{\mathcal{G}} \neg A_1 \approx \neg A_2 : -$$
$$\Theta \vDash_{\mathcal{G}} A_1 \otimes B_1 \approx A_2 \otimes B_2 : + \qquad \Theta \vDash_{\mathcal{G}} {}_s\Uparrow A_1 \approx {}_s\Uparrow A_2 : s$$

- *For any $\Theta \vDash_{\mathcal{G}} A_1 \approx A_2 : -$ and $\Theta \vDash_{\mathcal{G}} B_1 \approx B_2 : -$ we have*

$$\Theta \vDash_{\mathcal{G}} A_1 \,\&\, B_1 \approx A_2 \,\&\, B_2 : - \qquad \Theta \vDash_{\mathcal{G}} \ominus A_1 \approx \ominus A_1 : +$$
$$\Theta \vDash_{\mathcal{G}} A_1 \parr B_1 \approx A_2 \parr B_2 : - \qquad \Theta \vDash_{\mathcal{G}} {}_s\Downarrow A_1 \approx {}_s\Downarrow A_1 : s$$

- *For any $\Theta \vDash_{\mathcal{G}} A_1 \approx A_2 : s$ we have*

$$\Theta \vDash_{\mathcal{G}} \downarrow_s A_1 \approx \downarrow_s A_2 : + \qquad \Theta \vDash_{\mathcal{G}} \uparrow_s A_1 \approx \uparrow_s A_2 : -$$

- *For any $\Theta \vDash_{\mathcal{G}} F_1 \approx F_2 : s \to +$ and $\Theta \vDash_{\mathcal{G}} G_1 \approx G_2 : s \to -$ we have*

$$\Theta \vDash_{\mathcal{G}} \exists_s F_1 \approx \exists_s F_2 : + \qquad \Theta \vDash_{\mathcal{G}} \forall_s G_1 \approx \forall_s G_2 : -$$

Proof. The majority of these isomorphisms have been proven previously in [Dow17] (as a corollary of Theorem 8.8) for closed type isomorphisms, and they continue to hold exactly as before when generalized with the possibility of free type variables. Here, we show the key additional cases for the quantifiers $\forall$ and $\exists$, which make crucial use of the added parametricity of isomorphisms between open types. Since parametricity has been taken into account by definition, the isomorphisms themselves follow the standard form of unpacking structures built by the constructor of $\exists$ or $\forall$ in order to apply the given underlying isomorphism.

Consider the case for the existential quantifier $\exists_s$ (the case for $\forall_s$ is exactly dual), and assume that the given isomorphism $\Theta \vDash_{\mathcal{G}} F_1 \approx F_1 : s \to +$ is witnessed by

$$c_1 : (x_2 : F_2\ X \vdash_{\mathcal{G}}^{\Theta, X:s}\ \alpha_1 : F_1\ X) \qquad c_2 : (x_1 : F_1\ X \vdash_{\mathcal{G}}^{\Theta, X:s}\ \alpha_2 : F_2\ X)$$

We then have the following pair of commands c_1' and c_2' as witnesses to the isomorphism $\Theta \vDash_{\mathcal{G}} \exists_s F_1 \approx \exists_s F_2 : +$ (where we omit the $+$ annotation on all commands):

$$c_1' = \left\langle y_2 \middle| \exists_s F_2 \middle| \lambda(X \Cap x_2).\, \left\langle \mu\alpha_1.c_1 \middle| F_1 X \middle| \tilde{\mu}x_1.\, \langle X \Cap x_1 | \exists_s F_1 | \beta_1 \rangle \right\rangle \right\rangle : (y_2 : \exists_s F_2 \vdash_{\mathcal{G}}^{\Theta}\ \beta_1 : \exists_s F_1)$$
$$c_2' = \left\langle y_1 \middle| \exists_s F_1 \middle| \lambda(X \Cap x_1).\, \left\langle \mu\alpha_2.c_2 \middle| F_2 X \middle| \tilde{\mu}x_2.\, \langle X \Cap x_2 | \exists_s F_2 | \beta_2 \rangle \right\rangle \right\rangle : (y_1 : \exists_s F_1 \vdash_{\mathcal{G}}^{\Theta}\ \beta_2 : \exists_s F_2)$$

Notice how the private type X never escapes: it is existentially quantified in both the input (y_i) and output (β_j) (co)variables, and is only used by the process of the underlying isomorphism (c_1 or c_2). The compositions of c_1' and c_2' are an identity command as follows:

$$\begin{aligned}
&\left\langle \mu\beta_1.c_1' \middle| \exists_s F_1 \middle| \tilde{\mu}y_1.c_2' \right\rangle \\
&=_{\beta_\mu} \left\langle y_2 \middle| \exists_s F_2 \middle| \lambda(X \Cap x_2).\, \left\langle \mu\alpha_1.c_1 \middle| F_1 X \middle| \tilde{\mu}x_1.\, \left\langle X \Cap x_1 \middle| \exists_s F_1 \middle| \tilde{\mu}y_1.c_2' \right\rangle \right\rangle \right\rangle \\
&=_{\beta_{\tilde{\mu}}\beta_p} \left\langle y_2 \middle| \exists_s F_2 \middle| \lambda(X \Cap x_2).\, \left\langle \mu\alpha_1.c_1 \middle| F_1 X \middle| \tilde{\mu}x_1.\, \left\langle \mu\alpha_2.c_2 \middle| F_2 X \middle| \tilde{\mu}x_2.\, \langle X \Cap x_2 | \exists_s F_2 | \beta_2 \rangle \right\rangle \right\rangle \right\rangle \\
&=_{\chi^+} \left\langle y_2 \middle| \exists_s F_2 \middle| \lambda(X \Cap x_2).\, \left\langle \mu\alpha_2.\, \langle \mu\alpha_1.c_1 | F_1 X | \tilde{\mu}x_1.c_2 \rangle \middle| F_2 X \middle| \tilde{\mu}x_2.\, \langle X \Cap x_2 | \exists_s F_2 | \beta_2 \rangle \right\rangle \right\rangle \\
&=_{iso} \left\langle y_2 \middle| \exists_s F_2 \middle| \lambda(X \Cap x_2).\, \left\langle \mu\alpha_2.\, \langle x_2 | F_2 X | \alpha_2 \rangle \middle| F_2 X \middle| \tilde{\mu}x_2.\, \langle X \Cap x_2 | \exists_s F_2 | \beta_2 \rangle \right\rangle \right\rangle \\
&=_{\beta_\mu\beta_{\tilde{\mu}}} \left\langle y_2 \middle| \exists_s F_2 \middle| \lambda(X \Cap x_2).\, \langle X \Cap x_2 | \exists_s F_2 | \beta_2 \rangle \right\rangle \\
&=_{\eta_p} \langle y_2 | \exists_s F_2 | \beta_2 \rangle
\end{aligned}$$

where the other direction is the same, up to swapping the indexes 1 and 2. □

The goal now is to lift the compatibility of the core $\mathcal{D}$ connectives to any other context. This can be challenging with higher-order type variables. For example, suppose that $\vDash_{\mathcal{D}} A \approx B : s$. Why, then, is it the case that $X{:}s \to r \vDash_{\mathcal{D}} X\ A \approx X\ B : r$? The reason must have something to do with the higher-order variable X, but we know nothing about the type it will return. Therefore, we will rule out such cases, by requiring that types be concrete. This restriction is still more liberal than requiring that all types be closed, and is also limiting enough to prove compatibility under the binders of the quantifiers.

Definition 10.4 (Concrete Type)**.** A type A is *concrete* if A contains no subformula of the form $X\ A_1 \dots A_n$ where $n \geq 1$.

Lemma 10.5 (Isomorphism Substitution)**.** *For any concrete type* $\Theta, X : k \vdash_{\mathcal{D}} A : t... \to s$*, if* $\Theta \vDash_{\mathcal{D}} B \approx C : k$ *then* $\Theta \vDash_{\mathcal{D}} A[B/X] \approx A[C/X] : t... \to s$.

Proof. By induction on the derivation of $\Theta, X : k \vdash_{\mathcal{D}} A : t... \to s$ (where we assume that A is normalized without loss of generality):

- $A = X$ is immediate, by the definition of substitution.
- $A = Y$ (where $Y \neq X$) follows by reflexivity (Theorem 10.2).
- $A = \lambda Y{:}t_1.A'$ follows from the inductive hypothesis on the sub-derivation of the body $\Theta, Y : t_1, X : k \vdash_{\mathcal{D}} A' : t_2... \to s$ via the definition of higher-kinded type isomorphisms (Definition 10.1) and the fact that $(\lambda Y{:}t_1.A')\ Y =_\beta A'$.
- $A = \mathsf{F}\ A_1 \dots A_n$: follows from the inductive hypothesis on the sub-derivations of $A_1 \dots A_n$ via Theorem 10.3, since every core $\mathcal{D}$ connective has arguments of the appropriate kind (s or $t \to s$ for some s and t).

Note that it is not possible to have a case where $A = X\ A_1 \dots A_n$ (for a non-zero n), due to the assumption that A is concrete. □

Theorem 10.6 (Compatibility with Abstraction and Application)**.** *Parametric type isomorphism is compatible with type application and first-order type abstraction,* i.e., *for any concrete types* A, A' *and* $k = s... \to r$*:*

a) If $\Theta, X : t \vDash_{\mathcal{D}} A \approx A' : k$ *then* $\Theta \vDash_{\mathcal{D}} \lambda X{:}t.A \approx \lambda X{:}t.A' : t \to k$.
b) If $\Theta \vDash_{\mathcal{D}} A \approx A' : t \to k$ *and* $\Theta \vDash_{\mathcal{D}} B \approx B' : t$ *then* $\Theta \vDash_{\mathcal{D}} A\ B \approx A'\ B' : k$.

Proof. Part (a), which applies type isomorphisms underneath a type-level λ-abstraction, follows from the higher-order case of Definition 10.1. The conclusion is defined to be the same as $\Theta, X{:}t \vDash_{\mathcal{D}} (\lambda X{:}t.A)\ X \approx (\lambda X{:}t.A')\ X : s$ which follows from the assumption because $(\lambda X{:}t.A)\ X =_\beta A$ and $(\lambda X{:}t.A')\ X =_\beta A'$.

We will show part (b), which applies type isomorphisms onto the operation and operand of an application, in two parts. First, we focus on the operand. Note that the assumption that $\Theta \vDash_{\mathcal{D}} A \approx A' : t \to s... \to r$ is defined to be $\Theta, X{:}t, Y{:}s... \vDash_{\mathcal{D}} A\ X\ Y... \approx A'\ X\ Y... : r$ which comes with witnesses of the form

$$c : (x' : A'XY... \vdash_{\mathcal{D}}^{\Theta, X:t, Y:s...} \alpha : AXY...) \qquad c' : (x : AXY... \vdash_{\mathcal{D}}^{\Theta, X:t, Y:s...} \alpha' : A'XY...)$$

We also have the standard substitution property for typing derivations—if $c : (\Gamma \vdash_{\mathcal{D}}^{\Theta, X:k} \Delta)$ and $\Theta \vdash_{\mathcal{D}} A : k$ are derivable, then so is $c[A/X] : (\Gamma[A/X] \vdash_{\mathcal{D}}^{\Theta} \Delta[A/X])$—which follows by induction on the typing derivation of the command. Therefore, by substitution on the

witnesses to the isomorphism, we get

$$c[B/X] : (x' : A'BY... \vdash^{\Theta, Y:s...}_{\mathcal{D}} \alpha : ABY...) \quad c'[B/X] : (x : ABY... \vdash^{\Theta, Y:s...}_{\mathcal{D}} \alpha' : A'BY...)$$

so $\Theta, Y{:}s... \vDash_{\mathcal{D}} A\ B\ Y... \approx A'\ B\ Y... : r$, which is $\Theta \vDash_{\mathcal{D}} A\ B \approx A'\ B : k$, holds.

Next, note that by the previous Lemma 10.5, we can also perform the following substitution of $\Theta \vDash_{\mathcal{D}} B \approx B' : t$ into $\Theta, X{:}t \vdash_{\mathcal{D}} A'\ X : k$ to get $\Theta \vDash_{\mathcal{D}} A'\ B \approx A'\ B' : k$. Therefore by transitivity (Theorem 10.2) we get that $\Theta \vDash_{\mathcal{D}} A\ B \approx A'\ B' : k$. □

10.4. **Type isomorphisms are correct encodings.** We are now ready to put together the full isomorphism relating every type to its encoding. This is done incrementally by composing each of the base cases (that encoding a user-declared connective in terms of the core $\mathcal{D}$ connective) using the fact that isomorphism is a congruence relation.

Theorem 10.7 (Encoding Isomorphism)**.** *For all non-cyclic $\mathcal{G}$ extending $\mathcal{D}$ and concrete types $\Theta \vdash_{\mathcal{G}} A : k$, $\Theta \vDash_{\mathcal{G}} A \approx [\![A]\!]^{\mathcal{D}}_{\mathcal{G}} : k$.*

Proof. By induction on the derivation of $\Theta \vdash_{\mathcal{G}} A : k$, where we assume that A is normalized without loss of generality. Note that the encoded type has the same kind, but only using $\mathcal{D}$ connectives, *i.e.,* $\Theta \vdash_{\mathcal{D}} [\![A]\!]^{\mathcal{D}}_{\mathcal{G}} : k$. Therefore, we can apply compatibility (Theorem 10.6) in the case of an abstraction or application or reflexivity (Theorem 10.2) in the case of a type variable. The remaining cases are for the connectives F declared as data or codata types, which are given in Appendix A. □

But why is it useful to have the isomorphism $A \approx [\![A]\!]^{\mathcal{D}}_{\mathcal{G}}$? Because we can use it to automatically generate a local transformation between the two types! This is done by using the mappings of the isomorphism (which are represented internally as commands of the $\mathcal{CD}$ calculus) to selectively convert terms and coterms from one type to the other. And these transformations aren't arbitrary; they form an *equational correspondence* [SF93] which says that the exact same equalities hold between (co)terms of the two types. In other words, any optimization that was valid on the original type A is still valid after encoding it as $[\![A]\!]^{\mathcal{D}}_{\mathcal{G}}$ and vice versa.

Theorem 10.8 (Isomorphism Correspondence)**.** *If $\Theta \vDash_{\mathcal{G}} A \approx B : s$, then (co)terms of type A are in equational correspondence with (co)terms of type B, respectively.*

Proof. To establish an equational correspondence, we need to find maps between terms of the isomorphic types A and B. Since the same language serves as both the source and the target, we will use a pair of appropriate contexts based on the types A and B, that are derived from the given witnesses to the isomorphism, which can convert terms and coterms between types A and B. We can just focus on the case for converting terms, as coterm conversion is exactly dual.

Suppose that $c' : (x : A \vdash^{\Theta}_{\mathcal{G}} \beta : B)$ and $c : (y : B \vdash^{\Theta}_{\mathcal{G}} \alpha : A)$ witnesses the isomorphism $\Theta \vdash_{\mathcal{G}} A \approx B : s$. The desired contexts are then defined as $C = \mu\beta.\langle \square \big| A{:}s \big| \tilde{\mu}x.c' \rangle$ for converting a term from type A to B, and $C' = \mu\alpha.\langle \square | B{:}s | \tilde{\mu}y.c \rangle$ for converting back. To show that these context mappings give an equational correspondence, we must show that (1) the mappings preserve equality, and (2) both compositions of the mappings are an identity. The first fact follows immediately from compatibility; if two terms are equal, then they are equal in any context. The second fact is derived from the definition of type

isomorphisms at base kinds, which makes crucial use of χ^s equality (similar to Theorem 10.2). The composition for any term $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta$ (assuming $x, y \notin \Gamma$ and $\alpha, \beta \notin \Delta$) is:

$$\begin{aligned} C'[C[v]] &= \mu\alpha.\, \langle \mu\beta.\langle v | A{:}s | \tilde{\mu}x.c' \rangle \big| B{:}s \big| \tilde{\mu}y.c \rangle \\ &=_{\chi^s} \mu\alpha.\, \langle v \big| A{:}s \big| \tilde{\mu}x.\langle \mu\beta.c' | B{:}s | \tilde{\mu}y.c \rangle \rangle \\ &=_{iso} \mu\alpha.\, \langle v \big| A{:}s \big| \tilde{\mu}x.\, \langle x | A{:}s | \alpha \rangle \rangle \\ &=_{\eta_{\tilde{\mu}}\eta_{\mu}} v \end{aligned}$$

The reverse composition is also equal to the identity for any $\Gamma \vdash^{\Theta}_{\mathcal{G}} v' : B \mid \Delta$, where we have $\Gamma \vdash^{\Theta}_{\mathcal{G}} C[C'[v]] = v' : B \mid \Delta$. □

Corollary 10.9 ($\mathcal{D}$ Encoding Correspondence)**.** *For all non-cyclic $\mathcal{G}$ extending $\mathcal{D}$ and closed types $\vdash_{\mathcal{G}} A : s$, the (co)terms of type A are in equational correspondence with the (co)terms of type $[\![A]\!]^{\mathcal{D}}_{\mathcal{G}}$, respectively.*

This transformation acts as a local encoding that can be applied to just one subterm or sub-coterm of the entire program. In the field of compilers, this corresponds to an optimization referred to as "worker/wrapper." And the fact that it is an equational correspondence means that the rest cannot possibly tell the difference between the original expression and the encoded one, *i.e.,* this encoding faithfully reflects the extensional semantics in the programmer's definition.

11. Related Work

There have been several polarized languages [Lev01, Zei09, MM13], each with subtly different and incompatible restrictions on which programs are allowed to be written. The most common such restriction corresponds to *focusing* in logic [And92]; in the terms used here, focusing means that the parameters to constructors and observers *must* be values. Rather than impose a static focusing restriction on the syntax of programs, we instead imply a dynamic focusing behavior—which evaluates the parameters of constructors and observers before (co)pattern matching—during execution. Both static and dynamic notions of focusing are two sides of the same coin, and amount to the same end [DA18b].

The other restrictions vary between different frameworks, however. First, we might ask where computation can happen. In Levy's call-by-push-value [Lev01], which is based on the λ-calculus, value types (corresponding to positive types) can only be ascribed to values and computation can only occur at computation types (corresponding to negative types). In Zeilberger's calculus of unity [Zei08], which is based on the classical sequent calculus, isolates computation in a separate syntactic category of *statements* which do not have a return type, but is essentially the same as call-by-push-value in this regard as both frameworks only deal with *substitutable* entities, to the exclusion of named computations which may not be duplicated or deleted. Second, we might ask what are the allowable types for variables and, when applicable, covariables. In call-by-push-value, variables always have positive types, but in the calculus of unity variables have negative types or positive *atomic* types (and dually covariables have positive types or negative atomic types). These restrictions explain these two frameworks use a different conversion between the two different kinds of types: $\Uparrow$ introduces a positive variable and $\downarrow$ introduces a negative one, and in the setting of the sequent calculus $\Downarrow$ introduces a negative covariable and $\uparrow$ introduces a positive one. They also explain the calculus of unity's pattern matching: if there cannot be positive variables,

then pattern matching *must* continue until it reaches something non-decomposable like a λ-abstraction.

In contrast, System L allows for computation to occur at *any* type, and has no restrictions on the types of variables and covariables. In both of these ways, the System $\mathcal{D}$ is spiritually closest to System L. A motivation for this choice is that call-by-need forces more generality into the system: if there is no computation and no variables of call-by-need types, then the entire point of sharing work is missed. However, the call-by-value and -name sub-language can still be reduced down to the more restrictive style of call-by-push-value and the calculus of unity. We showed here that the two styles of positive and negative shifts are isomorphic, so the brunt of the work is to reduce to the appropriate normal form. Additionally, negative variables $x : A : -$ can be eliminated by substituting $y.\Downarrow$ for x where $y : \Downarrow A : +$, which satisfies the call-by-push-value restriction on variables. Alternatively, the calculus of unity restriction on (co)variables can be satisfied by type-directed η-expansion into nested (co)patterns.

Our notion of data and codata extends the "jumbo" connectives of Levy's jumbo λ-calculus [Lev06] to include a treatment of call-by-need as well the move from mono-discipline to multi-discipline. Our notion of (co)data is also similar to Zeilberger's [Zei09] definition of types via (co)patterns, which is fully dual, extended with sharing. This work also extends the work on the operational and equational theory of call-by-push-value [FSSS19] to incorporate not only "jumbo" connectives, but also call-by-need evaluation and its dual.

There has also been other recent work on extending call-by-push value with call-by-need evaluation [MM19], with the goal of studying the impact of computational effects on the differences between evaluation strategies. The main result of this work is to identify some effects which do not distinguish certain evaluation strategies. Similar to the fact that call-by-name and call-by-need evaluation give the same result when the only effect is non-termination [AMO$^+$95, AF97], McDermott and Mycroft showed how call-by-value and call-by-need evaluation give the same result with non-determinism as the only effect. As a methodology, [MM19] uses a call-by-push-value framework extended with a memoizing let construct similar to the one used here, but with the key difference that the types of call-by-need computations are the same as call-by-name ones. The tradeoff of this design difference is that there are fewer kinds of types (only value and computation types, as in call-by-push-value), but in turn types provide fewer guarantees (for instance, the type system does not ensure that call-by-need computations must be shared). As a consequence, it is not clear whether or not the type isomorphisms studied here—which rely heavily on the extensional properties which come from the connection between discipline and types—extend to this looser type system.

The challenges we faced in establishing a robust local encoding for user-defined (co)data types is similar to those encountered in the area of compositional compiler correctness [PA14, BSDA14, NHK$^+$15, SBCA15, GKR$^+$15, WWS19, PAC19, PA19]. Traditional compiler correctness only states that the compiled code exhibits the same behavior as the source program. As such, a compiler may only use a small fragment of the target language with a simpler semantics, and may use invariants between client and implementation code that are essential to its correctness. This rules out the possibility of linking with code in the target language that does not originate from that compiler; doing so may violate the compiler's invariants or go outside of its fragment of the target language. In contrast, compositional compiler correctness gives a much stronger guarantee that allows for safely linking with more code, whether it comes from a different compiler or is originally written in the target language. The difference between global compilation and local encodings illustrated in

Sections 9 and 10 is related to this difference between traditional and compositional compiler correctness, except that the encodings are performed *within* one language (here $\mathcal{CD}$) as opposed to *across* a source and target language of a compiler.

12. Conclusion

We have showed here how logical duality can help with compiling programs. On the one hand, the idea of polarity can be extended with other calling conventions like call-by-need and its dual, which opens up its applicability to the implementation of practical lazy functional languages. On the other hand, a handful of classical connectives can robustly compile and encode a wide variety of user-defined types, even for languages with effects, using the notion of parametric type isomorphisms. Parametricity lets us apply these encodings to common forms of type abstractions in programs, specifically parametric polymorphism and existentially-quantified abstract data types. Because of the duality of $\mathcal{D}$'s type abstractions $\forall$ and $\exists$, we can even encode some inductive *and* coinductive types and computations following the style of Böhm-Berarducci encodings in System F. Unfortunately, these encodings require too much indirection to be isomorphisms in the same sense as Section 10. To remedy this situation, the core $\mathcal{D}$ could also be extended with a form of type recursion capable of robustly encoding inductive and coinductive types, which we leave to future work.

As a next step, we intend to extend GHC's intermediate language, which already mixes call-by-need and call-by-value types, with the missing call-by-name types. Since it already has unboxed types [PJL91] corresponding to positive types, what remains are the fully extensional negative types. Crucially, we believe that negative function types would lift the idea of *call arity*—the number of arguments a function takes before "work" is done—from the level of terms to the level of types. Call arity is used to optimize curried function calls, since passing multiple arguments at once is more efficient that computing intermediate closures as each argument is passed one at a time. No work is done in a negative type until receiving an $\uparrow$ request or unpacking a $\Uparrow$ box, so polarized types compositionally specify multi-argument calling conventions.

For example, a binary function on integers would have the type $\mathsf{Int} \to \mathsf{Int} \to \uparrow \mathsf{Int}$, which only computes when both arguments are given, versus the type $\mathsf{Int} \to \uparrow\Downarrow(\mathsf{Int} \to \uparrow \mathsf{Int})$ which specifies work is done after the first argument, breaking the call into two steps since a closure must be evaluated and followed. This generalizes the existing treatment of function closures in call-by-push-value to call-by-need closures. The advantage of lifting this information into types is so that call arity can be taken advantage of in higher order functions. For example, the *zipWith* function takes a binary function to combine two lists, pointwise, and has the type $\forall X{:}\sharp.\forall Y{:}\sharp.\forall Z{:}\sharp.(X \to Y \to Z) \to [X] \to [Y] \to [Z]$. The body of *zipWith* does not know the call arity of the function it's given, but in the polarized type built with negative functions: $\forall X{:}\sharp.\forall Y{:}\sharp.\forall Z{:}\sharp.\Downarrow(\downarrow X \to \downarrow Y \to \uparrow Z) \to \downarrow[X] \to \downarrow[Y] \to \uparrow[Z]$ the interface in the type spells out that the higher-order function uses the faster two-argument calling convention.

Acknowledgments

This work is supported by the National Science Foundation under grants CCF-1719158 and CCF-1423617.

References

[ABS09] Zena M. Ariola, Aaron Bohannon, and Amr Sabry. Sequent calculi and abstract machines. *ACM Transactions on Programming Languages and Systems*, 31(4):13:1–13:48, May 2009.

[AF97] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997.

[AHS11] Zena M. Ariola, Hugo Herbelin, and Alexis Saurin. Classical call-by-need and duality. In *Typed Lambda Calculi and Applications: 10th International Conference*, TLCA'11, pages 27–44, Berlin, Heidelberg, June 2011. Springer Berlin Heidelberg.

[AMO+95] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246, New York, NY, USA, 1995. ACM.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992.

[BSDA14] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory c. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, pages 107–127, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[CH00] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 233–243, New York, NY, USA, 2000. ACM.

[DA14a] Paul Downen and Zena M. Ariola. Compositional semantics for composable continuations: From abortive to delimited control. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 109–122, New York, NY, USA, 2014. ACM.

[DA14b] Paul Downen and Zena M. Ariola. The duality of construction. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 249–269. Springer Berlin Heidelberg, Berlin, Heidelberg, April 2014.

[DA18a] Paul Downen and Zena M. Ariola. Beyond polarity: Towards a multi-discipline intermediate language with sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, pages 21:1–21:23, 2018.

[DA18b] Paul Downen and Zena M. Ariola. A tutorial on computational classical logic and the sequent calculus. *Journal of Functional Programming*, 28:e3, 2018.

[DMAPJ16] Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16, pages 74–88, New York, NY, USA, 2016. ACM.

[DMAV14] Paul Downen, Luke Maurer, Zena M. Ariola, and Daniele Varacca. Continuations, processes, and sharing. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, PPDP '14, pages 69–80, New York, NY, USA, 2014. ACM.

[Dow17] Paul Downen. *Sequent Calculus: A Logic and a Language for Computation and Duality*. PhD thesis, University of Oregon, 2017.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, 1991.

[FSSS19] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. Call-by-push-value in Coq: Operational, equational, and denotational theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 118–131, 2019.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. I. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[GKR+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, New York, NY, USA, 2015. ACM.

[Gri90] Timothy G. Griffin. A formulae-as-types notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[Hag87] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, pages 140–157, Berlin, Heidelberg, September 1987. Springer Berlin Heidelberg.

[Har12] Professor Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[Lev01] Paul Blain Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.

[Lev06] Paul Blain Levy. *Jumbo λ-Calculus*, pages 444–455. Springer Berlin Heidelberg, Berlin, Heidelberg, July 2006.

[MDAPJ17] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, pages 482–494, New York, NY, USA, June 2017. ACM.

[MM09] Guillaume Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic: 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL*, CSL 2009, pages 409–423, Berlin, Heidelberg, September 2009. Springer Berlin Heidelberg.

[MM13] Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot, 2013.

[MM14] Guillaume Munch-Maccagnoni. Formulae-as-types for an involutive negation. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 70:1–70:10, New York, NY, USA, 2014. ACM.

[MM19] Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In *Programming Languages and Systems*, pages 235–262, Cham, 2019. Springer International Publishing.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[NHK+15] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[PA14] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, pages 128–148, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[PA19] Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*, ICFP '19, New York, NY, USA, 2019. ACM.

[PAC19] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 51(6):125:1–125:36, February 2019.

[Par92] Michel Parigot. λμ-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning: International Conference*, LPAR '92, pages 190–201, Berlin, Heidelberg, July 1992. Springer Berlin Heidelberg.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.

[PJL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture: 5th ACM Conference*, pages 636–666, Berlin, Heidelberg, August 1991. Springer Berlin Heidelberg.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PM97] Simon Peyton Jones and Erik Meijer. Henk: a typed intermediate language. In *Proceedings of the First International Workshop on Types in Compilation*, 1997.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423, London, UK, UK, 1974. Springer-Verlag.

[SBCA15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 275–287, New York, NY, USA, 2015. ACM.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, November 1993.

[Wad03] Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 189–201, New York, NY, USA, 2003. ACM.

[Wad15] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, November 2015.

[WWS19] Yuting Wang, Pierre Wilke, and Zhong Shao. An abstract stack based approach to verified compositional compilation to machine code. *Proceedings of the ACM on Programming Languages*, 3(POPL):62:1–62:30, January 2019.

[Zei08] Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1):660–96, 2008.

[Zei09] Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

Appendix A. Isomorphisms for $\mathcal{D}$ Quantifiers and Encodings

A.1. **Isomorphism laws for type quantifiers.** Let's consider how to prove some of the specific isomorphism laws from Figure 24, namely the quantifier laws. Each law involving one of the quantifiers ($\forall$ or $\exists$) can be proved in terms of the notion of parametric type isomorphism, which means that there is *uniform* evidence that the type abstractions on both sides can be inter-converted without loss of information. The most important first step for establishing these isomorphisms between polarized connectives is to set up an isomorphism between their patterns or copatterns; the rest tends to follow. For our purposes here, we will focus on just the positive existential quantifier $\exists$, and note that the isomorphisms involving the negative universal quantifier $\forall$ are exactly dual to the following discussion.

The first isomorphism, $\exists X{:}s.\exists Y{:}t.A \approx \exists Y{:}t.\exists X{:}s.A$ swaps the type components between the patterns $X \mathbin{⅋} Y \mathbin{⅋} z$ and $Y \mathbin{⅋} X \mathbin{⅋} z$, and is effectively corresponds the commutation between the tuples (x, y) and (y, x). As such, when types are ignored the witnesses to the isomorphism is effectively the same (up to renaming bound type variables) in both direction, as only the type-level components are being swapped as follows:

$$c = \langle y|_+|\lambda(Y \mathbin{⅋} x).\langle x|_+|\lambda(X \mathbin{⅋} z).\langle X \mathbin{⅋} Y \mathbin{⅋} z|_+|\alpha\rangle\rangle : (y : \exists Y{:}t.\exists X{:}s.A \vdash_{\mathcal{D}} \alpha : \exists X{:}s.\exists Y{:}t.A)$$
$$c' = \langle x|_+|\lambda(X \mathbin{⅋} y).\langle y|_+|\lambda(Y \mathbin{⅋} z).\langle Y \mathbin{⅋} X \mathbin{⅋} z|_+|\beta\rangle\rangle : (x : \exists X{:}s.\exists Y{:}t.A \vdash_{\mathcal{D}} \beta : \exists Y{:}t.\exists X{:}s.A)$$

Notice that both compositions of these two commands are equal to the identities $\langle x|_+|\alpha\rangle$ and $\langle y|_+|\beta\rangle$, respectively, according to the β_μ, $\beta_{\tilde{\mu}}$, β_p, and finally η_p rules, in exact same manner that the commutation $A \otimes B \approx B \otimes A$ is derived. And since the above witnesses are uniform for any choice of type A, we can abstract away the specific A in the witnesses as

$$c : (y : \exists Y{:}t.\exists X{:}s.(F\ X\ Y) \vdash_{\mathcal{D}}^{F:s\to t\to +} \alpha : \exists X{:}s.\exists Y{:}t.(F\ X\ Y))$$
$$c' : (x : \exists X{:}s.\exists Y{:}t.(F\ X\ Y) \vdash_{\mathcal{D}}^{F:s\to t\to +} \beta : \exists Y{:}t.\exists X{:}s.(F\ X\ Y))$$

to get the stronger parametric isomorphism:

$$F : s \to t \to {}_+ \vDash_{\mathcal{D}} \exists X{:}s.\exists Y{:}t.(F\ X\ Y) \approx \exists Y{:}t.\exists X{:}s.(F\ X\ Y) : {}_+$$

Next, we have the isomorphism $\exists X{:}s.B \approx B$ where X is not free in B, just deletes the redundant abstraction which is never used. In the one direction we can insert an unneeded $X \mathbin{⅋} y$ around a value $x : B$, and in the other direction we can extract and return that B component (which is well-typed, because the existentially quantified variable X never escapes because B does not reference it), like so:

$$c' = \langle x|_+|\lambda X \mathbin{⅋} y.\langle y|_+|\beta\rangle\rangle \qquad : (x : \exists X{:}s.B \vdash_{\mathcal{D}} \beta : B)$$
$$c = \langle X \mathbin{⅋} y|_+|\alpha\rangle \qquad : (y : B \vdash_{\mathcal{D}} \alpha : \exists X{:}s.B)$$

One direction, which composes on the existential type $\exists X{:}s.B$, is just a straightforward calculation

$$\langle \mu\alpha.c|_+|\tilde{\mu}x.c'\rangle =_{\eta_{\tilde{\mu}}} \langle \mu\alpha.c|_+|\lambda X \mathbin{⅋} y.\langle y|_+|\beta\rangle\rangle =_{\beta_\mu} \langle X \mathbin{⅋} y|_+|\lambda X \mathbin{⅋} y.\langle y|_+|\beta\rangle\rangle =_{\beta_p} \langle y|_+|\beta\rangle$$

But the other direction, which composes on the arbitrary type $B : {}_+$, is equal to an identity due to the fact that the existential type B is call-by-value, which gives us the strongest possible β_μ rule in the following derivation:

$$\begin{aligned}\langle \mu\beta.c'|_+|\tilde{\mu}y.c\rangle &= \langle \mu\beta.c'|_+|\tilde{\mu}y.\langle X \mathbin{⅋} y|_+|\alpha\rangle\rangle \\ &=_{\beta_\mu} \langle x|_+|\lambda X \mathbin{⅋} y.\langle y|_+|\tilde{\mu}y.\langle X \mathbin{⅋} y|_+|\alpha\rangle\rangle\rangle\end{aligned}$$

$$=_{\beta_{\tilde{\mu}}} \langle x|_+|\lambda X \mathbin{⅋} y.\, \langle X \mathbin{⅋} y|_+|\alpha\rangle\rangle$$
$$=_{\eta_p} \langle x|_+|\alpha\rangle$$

And again, since the witness to the isomorphism is the same for any type $B : +$, we can abstract it out to get

$$c' : (x : \exists X{:}s.Y \vdash^{Y:+}_{\mathcal{D}} \beta : Y) \qquad\qquad c : (y : Y \vdash^{Y:+}_{\mathcal{D}} \alpha : \exists X{:}s.Y)$$

which guarantees the stronger parametric type isomorphism:

$$Y : + \vDash_{\mathcal{D}} \exists X{:}s.Y \approx Y : +$$

Finally, we have several isomorphisms for distributing other connectives over quantifiers. The most interesting ones are the ones involving negation

$$\ominus(\forall X{:}s.C) \approx \exists X{:}s.(\ominus C) \qquad\qquad \neg(\exists X{:}s.A) \approx \forall X{:}s.(\neg A)$$

since the second one holds in intuitionistic logic, but the first one does not. These isomorphisms relate the patterns $\ominus(X \mathbin{@} \beta) \approx X \mathbin{⅋} (\ominus\beta)$ for the first isomorphism and the copatterns $\neg[X \mathbin{⅋} y] \approx X \mathbin{@} [\neg y]$ for the second isomorphism. The second isomorphism can be translated to intuitionistic logic by encoding the negation $\neg A$ as $A \to \bot$, but the first isomorphism does not correspond to anything in intuitionistic logic. More formally, the witnesses to the first isomorphism are

$$c = \left\langle y \middle|_+ \middle| \lambda \ominus\delta.\, \langle \lambda[X \mathbin{@} \gamma].\, \langle X \mathbin{⅋} (\ominus\gamma)|_+|\alpha\rangle|_-|\delta\rangle \right\rangle : (y : \ominus(\forall X{:}s.C) \vdash_{\mathcal{D}} \alpha : \exists X{:}s.(\ominus C))$$
$$c' = \left\langle x \middle|_+ \middle| \lambda(X \mathbin{⅋} z).\, \langle z|_+|\lambda \ominus\gamma.\, \langle \ominus(X \mathbin{@} \gamma)|_+|\beta\rangle\rangle \right\rangle : (x : \exists X{:}s.(\ominus C) \vdash_{\mathcal{D}} \beta : \ominus(\forall X{:}s.C))$$

Both compositions of these mappings are equal to identity commands as follows, where η^-_q and η^+_p denote the full η laws (as derived in Section 7.3) that apply to any negative ($-$) term or positive ($+$) coterm, respectively:

$$\langle \mu\alpha.c|_+|\mu x.c'\rangle$$
$$=_{\beta_\mu} \left\langle y \middle|_+ \middle| \lambda \ominus\delta.\, \left\langle \lambda[X \mathbin{@} \gamma].\, \langle X \mathbin{⅋} (\ominus\gamma)|_+|\mu x.c'\rangle \middle|_- \middle| \delta \right\rangle \right\rangle$$
$$=_{\beta_{\tilde{\mu}}} \left\langle y \middle|_+ \middle| \lambda \ominus\delta.\, \left\langle \lambda[X \mathbin{@} \gamma].\, \left\langle X \mathbin{⅋} (\ominus\gamma) \middle|_+ \middle| \lambda(X \mathbin{⅋} z).\, \langle z|_+|\lambda \ominus\gamma.\, \langle \ominus(X \mathbin{@} \gamma)|_+|\beta\rangle\rangle \right\rangle \middle|_- \middle| \delta \right\rangle \right\rangle$$
$$=_{\beta_p} \left\langle y \middle|_+ \middle| \lambda \ominus\delta.\, \left\langle \lambda[X \mathbin{@} \gamma].\, \langle \ominus\gamma|_+|\lambda \ominus\gamma.\, \langle \ominus(X \mathbin{@} \gamma)|_+|\beta\rangle\rangle \middle|_- \middle| \delta \right\rangle \right\rangle$$
$$=_{\beta_p} \left\langle y \middle|_+ \middle| \lambda \ominus\delta.\, \langle \lambda[X \mathbin{@} \gamma].\, \langle \ominus(X \mathbin{@} \gamma)|_+|\beta\rangle|_-|\delta\rangle \right\rangle$$
$$=_{\beta_{\tilde{\mu}}} \left\langle y \middle|_+ \middle| \lambda \ominus\delta.\, \langle \lambda[X \mathbin{@} \gamma].\, \langle \mu\alpha.\, \langle \ominus\alpha|_+|\beta\rangle|_-|X \mathbin{@} \gamma\rangle|_-|\delta\rangle \right\rangle$$
$$=_{\eta^-_q} \langle y|_+|\lambda \ominus\delta.\, \langle \mu\alpha.\, \langle \ominus\alpha|_+|\beta\rangle|_-|\delta\rangle\rangle$$
$$=_{\beta_\mu} \langle y|_+|\lambda \ominus\delta.\, \langle \ominus\delta|_+|\beta\rangle\rangle$$
$$=_{\eta_p} \langle y|_+|\beta\rangle$$

$$\langle \mu\beta.c'|_+|\tilde{\mu}y.c\rangle$$
$$=_{\beta_\mu} \left\langle x \middle|_+ \middle| \lambda(X \mathbin{⅋} z).\, \langle z|_+|\lambda \ominus\gamma.\, \langle \ominus(X \mathbin{@} \gamma)|_+|\tilde{\mu}y.c\rangle\rangle \right\rangle$$

$$
\begin{aligned}
&=_{\beta_{\tilde{\mu}}} \left\langle x \middle|_+ \middle| \curlywedge(X \mathbin{⅋} z).\, \left\langle z \middle|_+ \middle| \curlywedge\ominus\gamma.\, \left\langle \ominus(X \mathbin{@} \gamma) \middle|_+ \middle| \curlywedge\ominus\delta.\, \langle \lambda[X \mathbin{@} \gamma].\, \langle X \mathbin{⅋} (\ominus\gamma) |_+ |\alpha\rangle |_- |\delta\rangle \right\rangle \right\rangle \right\rangle \\
&=_{\beta_p} \left\langle x \middle|_+ \middle| \curlywedge(X \mathbin{⅋} z).\, \left\langle z \middle|_+ \middle| \curlywedge\ominus\gamma.\, \langle \lambda[X \mathbin{@} \gamma].\, \langle X \mathbin{⅋} (\ominus\gamma) |_+ |\alpha\rangle |_- |X \mathbin{@} \gamma\rangle \right\rangle \right\rangle \\
&=_{\beta_q} \left\langle x \middle|_+ \middle| \curlywedge(X \mathbin{⅋} z).\, \langle z |_+ |\curlywedge\ominus\gamma.\, \langle X \mathbin{⅋} (\ominus\gamma) |_+ |\alpha\rangle \rangle \right\rangle \\
&=_{\beta_{\tilde{\mu}}} \left\langle x \middle|_+ \middle| \curlywedge(X \mathbin{⅋} z).\, \langle z |_+ |\curlywedge\ominus\gamma.\, \langle \ominus\gamma |_+ |\tilde{\mu}y.\, \langle X \mathbin{⅋} y |_+ |\alpha\rangle \rangle \rangle \right\rangle \\
&=_{\eta_p^+} \langle x |_+ |\curlywedge(X \mathbin{⅋} z).\, \langle z |_+ |\tilde{\mu}y.\, \langle X \mathbin{⅋} y |_+ |\alpha\rangle \rangle \rangle \\
&=_{\beta_{\tilde{\mu}}} \langle x |_+ |\curlywedge(X \mathbin{⅋} z).\, \langle X \mathbin{⅋} y |_+ |\alpha\rangle \rangle \\
&=_{\eta_p} \langle x |_+ |\alpha\rangle
\end{aligned}
$$

Notice how the interplay between both call-by-value and call-by-name semantics as used in the two quantifiers and negation connectives, and its impact on ensuring the full η laws, are essential for deriving the above isomorphism.

The other isomorphisms for distributing multiplicative connectives over quantifiers given in Figure 24 can be witnessed in the same way by the following relationship between (co)patterns that reassociate the two different forms of grouping operations:

$$((X \mathbin{⅋} x), y) \approx X \mathbin{⅋} (x, y) \qquad\qquad [[X \mathbin{@} \alpha], \beta] \approx X \mathbin{@} [\alpha, \beta]$$

However, notice the conspicuously missing laws for distributing the additive connectives over their matching quantifiers:

$$(\forall X{:}s.C) \mathbin{\&} D \approx \forall X{:}s.(C \mathbin{\&} D) \qquad\qquad (\exists X{:}s.A) \oplus B \approx \exists X{:}s.(A \oplus B)$$

That's because there is some missing information (the instantiation choice for the quantified type X) that is unavailable when the other constructor or destructor (containing B or D, respectively) is taken. For example, if we were to try to set up a correspondence between the patterns of the second isomorphism, the best we can do is as follows:

$$\begin{aligned} \iota_1(X \mathbin{⅋} x) &\Rightarrow X \mathbin{⅋} (\iota_1 x) & X \mathbin{⅋} (\iota_1 x) &\Rightarrow \iota_1(X \mathbin{⅋} x) \\ \iota_2 y &\Rightarrow ({}_s\Uparrow 0) \mathbin{⅋} (\iota_2 y) & X \mathbin{⅋} (\iota_2 y) &\Rightarrow \iota_2 y \end{aligned}$$

Notice how there is no quantified type given in the pattern $\iota_2 y$, so we can just use an arbitrary type like ${}_s\Uparrow 0$ since it is not referenced by y, anyway. However, the consequence of picking an arbitrary type is that a round-trip sends the pattern $X \mathbin{⅋} (\iota_2 y)$ to $({}_s\Uparrow 0) \mathbin{⅋} (\iota_2 y)$. The same thing happens if we try to set up a correspondence between the copatterns of the first isomorphism like so

$$\begin{aligned} \pi_1[X \mathbin{@} \alpha] &\Rightarrow X \mathbin{@} [\pi_1 \alpha] & X \mathbin{@} [\pi_1 \alpha] &\Rightarrow \pi_1[X \mathbin{@} \alpha] \\ \pi_2 \beta &\Rightarrow [{}_s\Downarrow \top] \mathbin{@} [\pi_2 \beta] & X \mathbin{@} [\pi_2 \beta] &\Rightarrow \pi_2 \beta \end{aligned}$$

where a round-trip sends the copattern $X \mathbin{@} [\pi_2 \beta]$ to $[{}_s\Downarrow \top] \mathbin{@} [\pi_2 \beta]$.

But this problem is somewhat artificial: the semantics we give in Figures 18 and 19 ignores these type annotations, so they play no part in the dynamic interpretation of programs. This fact means that we could enhance the η laws for quantifiers to actually ignore the chosen types as follows:

$$\lambda[X \mathbin{@} \beta].\, \langle x | A{:}_- | B \mathbin{@} \beta \rangle = x \qquad\qquad \curlywedge(X \mathbin{⅋} y).\, \langle B \mathbin{⅋} y | A{:}_+ | \alpha \rangle = \alpha$$

These equations would be sound as long as we fully commit to a parametric and type-irrelevant semantics for the quantifiers. This can be done formally by means of type erasure,

a parametric reducibility candidate semantics, or just ignoring type annotations in the results of programs. And the enhanced η laws above are enough to prove that the additive connectives ($\oplus$ and $\&$) distribute over the quantifiers ($\exists$ and $\forall$, respectively). Intuitively, this fact states that the choice of the quantified type (chosen by the producer for $\exists$ and chosen by the consumer for $\forall$) cannot impact other branches of computation where it is not used. For example, a producer of type $\forall X{:}s.(C \,\&\, D)$, where X is not referenced in D, must produce the exact same result when asked for its second component no matter what type the consumer chooses for X.

A.2. **Equational reasoning for macro (co)patterns.** The encodings of declared data and codata types involves a macro-expansion of flat (co)patterns (of the form $\mathsf{K}\,X...\alpha...y...$ and $\mathsf{O}\,X...y...\alpha...$) into nested (co)patterns built from the constructors and destructors of the core $\mathcal{D}$ types. Thankfully, these encodings follow a very particular form, where the *additive* (co)patterns (those with multiple options like ι_i of the $\oplus$ connective and $\pi_1 i$ of the $\&$ connective) are never found inside the *multiplicative* (co)patterns (those with multiple sub-patterns like (p_1, p_2) of the $\otimes$ connective and $[q_1, q_2]$ of the $\parr$ connective). This special property makes macro-expanding patterns and copatterns rather straightforward, which can be done with the following equations:

$$\begin{aligned}
\lambda\{_s{\Downarrow}q.c...\} &= \lambda\big\{_s{\Downarrow}\alpha.\big\langle\lambda\{q.c...\}\big|_-\big|\alpha\big\rangle\big\} & \text{\reflectbox{λ}}\{\,_s{\Uparrow}p.c...\} &= \text{\reflectbox{λ}}\big\{\,_s{\Uparrow}x.\big\langle x\big|_+\big|\text{\reflectbox{λ}}\{p.c...\}\big\rangle\big\} \\
\lambda\begin{Bmatrix}\pi_1 q_1.c_1... \\ \pi_2 q_2.c_2...\end{Bmatrix} &= \lambda\begin{Bmatrix}\pi_1\alpha.\big\langle\lambda\{q_1.c_1...\}\big|_-\big|\alpha\big\rangle \\ \pi_2\beta.\big\langle\lambda\{q_2.c_2...\}\big|_-\big|\beta\big\rangle\end{Bmatrix} & \text{\reflectbox{λ}}\begin{Bmatrix}\iota_1 p_1.c_1... \\ \iota_2 p_2.c_2...\end{Bmatrix} &= \text{\reflectbox{λ}}\begin{Bmatrix}\iota_1 x.\big\langle x\big|_+\big|\text{\reflectbox{λ}}\{p_1.c_1...\}\big\rangle \\ \iota_2 y.\big\langle y\big|_+\big|\text{\reflectbox{λ}}\{p_2.c_2...\}\big\rangle\end{Bmatrix} \\
\lambda[X \mathbin{@} q].c &= \lambda[X \mathbin{@} \alpha].\,\langle\lambda q.c|_-|\alpha\rangle & \text{\reflectbox{λ}}(X \mathbin{\mathbb{O}} p).c &= \text{\reflectbox{λ}}(X \mathbin{\mathbb{O}} y).\,\langle y|_+|\text{\reflectbox{λ}}p.c\rangle \\
\lambda[q_1, q_2].c &= \lambda[\alpha,\beta].\,\big\langle\lambda q_1.\,\langle\lambda q_2.c|_-|\beta\rangle\big|_-\big|\alpha\big\rangle & \text{\reflectbox{λ}}(p_1, p_2).c &= \text{\reflectbox{λ}}(x, y).\,\big\langle x\big|_+\big|\text{\reflectbox{λ}}p_1.\,\langle y|_+|\text{\reflectbox{λ}}p_2.c\rangle\big\rangle \\
\lambda[\neg p].c &= \lambda[\neg x].\langle x|_+|\text{\reflectbox{λ}}p.c\rangle & \text{\reflectbox{λ}}(\ominus q).c &= \text{\reflectbox{λ}}(\ominus\alpha).\langle\text{\reflectbox{λ}}p.c|_-|\alpha\rangle \\
\lambda\alpha.c &= \mu\alpha.c & \text{\reflectbox{λ}}x.c &= \tilde{\mu}x.c
\end{aligned}$$

Using this macro expansion, we can verify that the correspondingly expanded pattern-matching equalities can be derived from the simpler rules of the core $\mathcal{D}$ equational theory.

Lemma A.1 (Macro (co)pattern matching)**.** *Each of the following operational steps and equations, given by the macro-expansion* $[\![-]\!]^{\mathcal{D}}_{\mathcal{G}}$*:*

$$\begin{aligned}
&([\![\beta_q]\!]^{\mathcal{D}}_{\mathcal{G}}) && \Big\langle\lambda\{\cdots \mid [\![q]\!]^{\mathcal{D}}_{\mathcal{G}}.c \mid \ldots\}\Big|s\Big|[\![q]\!]^{\mathcal{D}}_{\mathcal{G}}[\rho]\Big\rangle \mapsto\!\!\!\!\twoheadrightarrow c[\rho] \\
&([\![\beta_p]\!]^{\mathcal{D}}_{\mathcal{G}}) && \Big\langle[\![p]\!]^{\mathcal{D}}_{\mathcal{G}}[\rho]\Big|s\Big|\text{\reflectbox{λ}}\{\cdots \mid [\![p]\!]^{\mathcal{D}}_{\mathcal{G}}.c \mid \ldots\}\Big\rangle \mapsto\!\!\!\!\twoheadrightarrow c[\rho] \\
&([\![\eta_q]\!]^{\mathcal{D}}_{\mathcal{G}}) && \lambda\Big\{[\![q]\!]^{\mathcal{D}}_{\mathcal{G}}.\langle x|s|[\![q]\!]^{\mathcal{D}}_{\mathcal{G}}\rangle...\Big\} = x \\
&([\![\eta_p]\!]^{\mathcal{D}}_{\mathcal{G}}) && \text{\reflectbox{λ}}\Big\{[\![p]\!]^{\mathcal{D}}_{\mathcal{G}}.\langle[\![p]\!]^{\mathcal{D}}_{\mathcal{G}}|s|\alpha\rangle...\Big\} = \alpha
\end{aligned}$$

are derivable in the core $\mathcal{D}$*-calculus, where* $[\![\beta_p]\!]^{\mathcal{D}}_{\mathcal{G}}$ *and* $[\![\beta_p]\!]^{\mathcal{D}}_{\mathcal{G}}$ *take* one or more *steps.*

Proof. The macro $[\![\beta_p]\!]^{\mathcal{D}}_{\mathcal{G}}$ and $[\![\beta_q]\!]^{\mathcal{D}}_{\mathcal{G}}$ reductions can be calculated from the small-step β_p and β_q rules given in Figure 18. Likewise, the macro $[\![\eta_p]\!]^{\mathcal{D}}_{\mathcal{G}}$ and $[\![\eta_q]\!]^{\mathcal{D}}_{\mathcal{G}}$ reductions can be calculated from the small-step η_p and η_q rules with the help of $\beta_\mu\beta_{\tilde{\mu}}\eta_\mu\eta_{\tilde{\mu}}$ to apply to

(co)values (and not just (co)variables) as was shown in Section 7. The calculations follow the same structure as in [Dow17] Theorem 8.1, but with the addition of type-quantifying patterns for $\forall$ and $\exists$ that are analogous to simpler cases for $\otimes$ and $\parr$ (co)patterns. □

A.3. **Isomorphisms between declared types and their encoding.**

Lemma A.2 (Data Encoding Isomorphism)**.** *Given any $\mathcal{G}$ extending $\mathcal{D}$, and any data type declaration*

$$\begin{aligned}&\textbf{data}\,\mathsf{F}(X:k)...:s\,\textbf{where}\\&\quad \mathsf{K}_1:\left(A_1:t_1... \vdash^{Y_1:s_1...}\mathsf{F}\,X... \mid B_1:r_1...\right)\\&\quad \mathsf{K}_n:\left(A_n:t_1... \vdash^{Y_n:s_n...}\mathsf{F}\,X... \mid B_n:r_n...\right)\end{aligned}$$

in $\mathcal{G}$, it follows that $\vDash \mathsf{F} \approx [\![\mathsf{F}]\!]^{\mathcal{D}}_{\mathcal{G}} : k... \to s$.

Proof. Given the patterns $p_i = \mathsf{K}_i\, Y_i...\alpha_i...x_i...$ matching the above declaration for i ranging from 1 to n, the witness to the isomorphism is given by the commands:

$$\begin{aligned}c &= \left\langle x' \middle| s \middle| \curlywedge\left\{ [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}.\langle p_i|s|\alpha\rangle... \right\}\right\rangle : (x' : [\![\mathsf{F}]\!]^{\mathcal{D}}_{\mathcal{G}}\,X... \vdash^{X:k...}_{\mathcal{G}} \alpha : \mathsf{F}\,X...)\\ c' &= \left\langle x \middle| s \middle| \curlywedge\left\{ p_i.\langle [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}|s|\alpha'\rangle... \right\}\right\rangle : (x : \mathsf{F}\,X... \vdash^{X:k...}_{\mathcal{G}} \alpha' : [\![\mathsf{F}]\!]^{\mathcal{D}}_{\mathcal{G}}X...)\end{aligned}$$

Both compositions are equal to the identity mapping. For the identity on F, we have

$$\begin{aligned}\left\langle \mu\alpha'.c' \middle| s \middle| \tilde{\mu}x'.c \right\rangle &=_{\eta_{\tilde{\mu}}} \left\langle \mu\alpha'.c' \middle\| \curlywedge\left\{ p_i.\langle [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}|s|\alpha'\rangle... \right\}\right\rangle\\ &=_{\beta_\mu} \left\langle x \middle| s \middle| \curlywedge\left\{ p_i.\left\langle [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}} \middle| s \middle| \curlywedge\{ [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}.\langle p_i|s|\alpha\rangle... \}\right\rangle... \right\}\right\rangle\\ &=_{[\![\beta_p]\!]^{\mathcal{D}}_{\mathcal{G}}} \left\langle x \middle| s \middle| \curlywedge\left\{ p_i.\langle [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}|s|\alpha'\rangle... \right\}\right\rangle\\ &=_{\eta_p} \langle x|s|\alpha\rangle\end{aligned}$$

And for the identity on the encoding $[\![\mathsf{F}]\!]^{\mathcal{D}}_{\mathcal{G}}$, we have

$$\begin{aligned}\left\langle \mu\alpha.c \middle| s \middle| \tilde{\mu}x.c' \right\rangle &=_{\eta_{\tilde{\mu}}} \left\langle \mu\alpha.c \middle| s \middle| \curlywedge\left\{ p_i.\langle [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}|s|\alpha'\rangle... \right\}\right\rangle\\ &=_{\beta_\mu} \left\langle x' \middle| s \middle| \curlywedge\left\{ [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}.\left\langle p_i \middle| s \middle| \curlywedge\{ p_i.\langle [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}|s|\alpha'\rangle... \}\right\rangle... \right\}\right\rangle\\ &=_{\beta_p} \left\langle x' \middle| s \middle| \curlywedge\left\{ [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}.\langle [\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}|s|\alpha'\rangle... \right\}\right\rangle\\ &=_{[\![\eta_p]\!]^{\mathcal{D}}_{\mathcal{G}}} \left\langle x' \middle| s \middle| \alpha' \right\rangle\end{aligned}$$

□

Lemma A.3 (Codata Encoding Isomorphism)**.** *Given any $\mathcal{G}$ extending $\mathcal{D}$, and any codata type declaration*

$$\begin{aligned}&\textbf{codata}\,\mathsf{G}(X:k)...:s\,\textbf{where}\\&\quad \mathsf{O}_1:\left(A_1:t_1... \mid \mathsf{G}\,X... \vdash^{Y_1:s_1...} B_1:r_1...\right)\\&\quad \mathsf{O}_n:\left(A_n:t_1... \mid \mathsf{G}\,X... \vdash^{Y_n:s_n...} B_n:r_n...\right)\end{aligned}$$

in $\mathcal{G}$, it follows that $\vDash \mathsf{G} \approx [\![\mathsf{G}]\!]^{\mathcal{D}}_{\mathcal{G}} : k... \to s$.

Proof. Dual to the proof of Lemma A.2. □

Appendix B. Type Safety

A side effect of disciplines linking the static and dynamic semantics is that the operational semantics is defined over *typed* (open) commands. However, there is a convenient way to generalize the operational semantics to work with untyped terms as well. In particular, the types in variable binding annotations are irrelevant for computation; only the discipline symbol s has any impact on evaluation. An easy way to remove the restriction of typing is to collapse types, which we can do by adding the following rule to the kind system:

$$\frac{\Theta \vdash_{\mathcal{G}} A : s \quad \Theta \vdash_{\mathcal{G}} B : s}{\Theta \vdash_{\mathcal{G}} A = B : s}\ \textit{Untype}$$

so that for any two types A and B of kind s and term v, we have $v : A$ if and only if $v : B$, and similarly for coterms. Therefore, there is no longer any reason to keep track of types, only disciplines, so we may as well write commands as $\langle v|s|e\rangle$ instead of $\langle v|A{:}s|e\rangle$.

Definition B.1 (Well-disciplined)**.** The *discipline system* of $\mathcal{D}$ is the type system of $\mathcal{D}$ extended with the *Untype*, and we say that commands, terms, and coterms are *well-disciplined* when they have some derivation in the discipline system.

The discipline system accepts strictly more expressions than the type system, since it expresses general recursion in pure $\mathcal{D}$ which is not well-typed. For example, an analogue to the usual non-terminating λ-calculus term $(\lambda x.xx)\ (\lambda x.xx)$ can be written as the following well-disciplined $\mathcal{D}$ command using negation and the shift $\downarrow$ for converting the negative value x to the positive value $\downarrow x$:

$$\left\langle \lambda\{\neg\downarrow x.\langle x|-|\neg\downarrow x\rangle\}\middle|-\middle|\tilde{\mu}y.\,\langle y|-|\neg\downarrow y\rangle\right\rangle$$

Similarly, a fixed-point combinator corresponding to $\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$ can be expressed in the well-disciplined calculus as follows, where we write $W \cdot F$ as shorthand for $[\neg W, F]$:

$$\lambda[\downarrow f \cdot \alpha].\left\langle \lambda[\downarrow x \cdot \beta].\left\langle f\middle|-\middle|\downarrow\mu\gamma.\,\langle x|-|\downarrow x \cdot \gamma\rangle \cdot \beta\right\rangle\middle|-\middle|\tilde{\mu}g.\,\langle g|-|\downarrow g \cdot \alpha\rangle\right\rangle$$

This approach to untyped multi-discipline semantics, via relaxing a traditional type system, follows from [Zei09]. The discipline system can also be captured directly via the grammar of terms itself in the style of [Lev01] and [MM13], which demonstrates that disciplines can be easily inferred from the syntax of terms. So in this sense, the operational semantics in Figure 19 can also be used as a semantics for untyped, but well-disciplined, commands.

As a notational convenience, we say that $c \not\mapsto$ if there is no c' such that $c \not\mapsto c'$ (and dually $c \mapsto$ if there is a c' such that $c \mapsto c'$). The first basic property of call-by-(co)need evaluation is that (co)values do not have any standard reduction, and symmetrically, any μ- or $\tilde{\mu}$-abstraction whose body has a standard reduction cannot be a (co)value.

Lemma B.2 ((Co)Need Normal Forms)**.**
a) If $\tilde{\mu}x.c$ is a $\sharp$ covalue then $c \not\mapsto$.
b) If $\mu\alpha.c$ is a $\flat$ value then $c \not\mapsto$.
c) If H does not bind x then $H[\langle x|A{:}\sharp|E_\sharp\rangle] \not\mapsto$.
d) If H does not bind α then $H[\langle V_\flat|A{:}\flat|\alpha\rangle] \not\mapsto$.

Proof. (a) and (c) both follow simultaneously by induction on the definition of $\sharp$ covalues and heaps. The interesting case is when we have $\left\langle x\middle|A{:}\sharp\middle|\tilde{\mu}y.H[\langle y|B{:}\sharp|E_\sharp\rangle]\right\rangle$ where H does not bind y: the $\beta^{\pm}_{\tilde{\mu}}$ rule doesn't apply to the top-level command (because $\sharp \notin \{+, -, \flat\}$), neither does the $\phi^{\sharp}_{\tilde{\mu}}$ rule (because x is a variable), and furthermore $H[\langle y|B{:}\sharp|E_\sharp\rangle] \not\mapsto$ by the

inductive hypothesis. Likewise, (b) and (d) both follow simultaneously by induction on the definition of $\flat$ values and heaps. □

Corollary B.3 ((Co)Need Non-Normal Forms)**.** *If $c \mapsto c'$ then*

a) $\tilde{\mu}x.c$ is not a $\sharp$ covalue, and
b) $\mu\alpha.c$ is not a $\flat$ value.

Corollary B.4 (Needed $\mu\tilde{\mu}$-abstractions)**.** *a) If $\tilde{\mu}x.c$ is a $\sharp$ covalue, then $x \in \mathrm{NV}(c)$.*
b) If $\mu\alpha.c$ is a $\flat$ value, then $\alpha \in \mathrm{NV}(c)$.

Note that there may be some μ- or $\tilde{\mu}$-abstractions that are not (co)values, but also cannot reduce. For example, consider $\tilde{\mu}x.\langle y|A{:}\sharp|\alpha\rangle$: this coterm is not a covalue because it ignores its input, but it needs to know the value of y in order to progress. However, the above two properties are enough to guarantee that standard reduction is always deterministic, even with call-by-(co)need evaluation steps.

Lemma B.5 (Determinism)**.** *If $c_1 \leftarrow\!\shortmid\, c \mapsto c_2$ then c_1 and c_2 are the same command.*

Proof. First, we show the base case when a reduction is applied to the top-level of a command, by induction on the syntax of c. Note that for commands in which the $\beta^{\pm}_{\mu}$, $\beta^{\pm}_{\tilde{\mu}}$, β_p, or β_q rules apply directly, determinism is immediate because no other rules apply to the top of the command, and the only context H such that $c = H[c']$ is the empty context. The only remaining cases are for the $\phi^{\sharp}_{\tilde{\mu}}$ and $\phi^{\flat}_{\mu}$ rules:

- $(\phi^{\sharp}_{\tilde{\mu}})$ $\langle V_{\sharp}|A{:}\sharp|\tilde{\mu}x.c\rangle \mapsto c[V_{\sharp}/x{::}\sharp]$ where $\tilde{\mu}x.c$ is a $\sharp$ covalue. Note that $c \not\mapsto$ due to Lemma B.2, so this $\phi^{\sharp}_{\tilde{\mu}}$ reduction is the only one that applies.
- $(\phi^{\flat}_{\mu})$ follows dually to the above case.

Next, we must consider when a reduction is applied inside a heap, so that $c = H[c']$ and $c'_1 \leftarrow\!\shortmid\, c' \mapsto c'_2$, which follows by induction on H. The base case is shown above, and the two remaining cases are for delayed bindings:

- Given $c = \langle v | A{:}\sharp | \tilde{\mu}x.H'[c'] \rangle$, then we know that $\tilde{\mu}x.H'[c']$ is not a $\sharp$ covalue (Corollary B.3), so $\phi^{\sharp}_{\tilde{\mu}}$ cannot fire to the top of this command, and the result follows by the inductive hypothesis.
- The dual case where $c = \langle \mu\alpha.H'[c'] \| A{:}\flat \rangle\, e$ follows dually to the above. □

To demonstrate the closure of standard reduction under substitution, we need a more formal way to determine that a substitution is well-disciplined or well-typed. The definition of the typing rules for substitutions, of the form $(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta) \Rightarrow \rho : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta')$ where ρ transforms the sequent on the left in order the one on the right, is:

$$\frac{}{(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta) \Rightarrow \varepsilon : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta)}$$

$$\frac{(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta) \Rightarrow \rho : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta') \quad \Theta' \vdash_{\mathcal{G}} A[\rho] : s \quad \Gamma' \vdash^{\Theta'}_{\mathcal{G}} V_s : A[\rho]\ ;\ \Delta'}{(\Gamma, x : A \vdash^{\Theta}_{\mathcal{G}} \Delta) \Rightarrow \rho[V_s/x{::}s] : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta')}$$

$$\frac{(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta) \Rightarrow \rho : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta') \quad \Theta' \vdash_{\mathcal{G}} A[\rho] : s \quad \Gamma'\ ;\ E_s : A[\rho] \vdash^{\Theta'}_{\mathcal{G}} \Delta'}{(\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A, \Delta) \Rightarrow \rho[E_s/\alpha{::}s] : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta')}$$

$$\frac{\Theta' \vdash_{\mathcal{G}} A : k \quad (\Gamma[A/X{:}k] \vdash^{\Theta}_{\mathcal{G}} \Delta[A/X{:}k]) \Rightarrow \rho : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta')}{(\Gamma \vdash^{\Theta, X:k}_{\mathcal{G}} \Delta) \Rightarrow \rho[A/X{:}k] : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta')}$$

As with typing expressions (commands, terms, and coterms), we can weaken the above rules to only checking that a substitution is well-disciplined. A substitution ρ is well-typed is well-disciplined if it has a typing derivation possibly using the *Untype* rule from Definition B.1. With this typing and discipline system for substitutions, we can see that (1) typing derivations, (2) the syntax of (co)values, and (3) the reductions of the operational semantics are all closed under a matching substitution.

Lemma B.6 (Typing Substitution Closure)**.** *Let* $(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta) \Rightarrow \rho : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta')$.

a) If $\Theta \vdash_{\mathcal{G}} A : k$ *then* $\Theta' \vdash_{\mathcal{G}} A[\rho] : k$.
b) If $c : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta)$ *then* $c[\rho] : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta')$.
c) If $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A \mid \Delta$ *then* $\Gamma' \vdash^{\Theta'}_{\mathcal{G}} v[\rho] : A[\rho] \mid \Delta'$.
d) If $\Gamma \vdash^{\Theta}_{\mathcal{G}} v : A ; \Delta$ *then* $\Gamma' \vdash^{\Theta'}_{\mathcal{G}} v[\rho] : A[\rho] ; \Delta'$.
e) If $\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ *then* $\Gamma' \mid e[\rho] : A[\rho] \vdash^{\Theta'}_{\mathcal{G}} \Delta'$.
f) If $\Gamma ; e : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ *then* $\Gamma' ; e[\rho] : A[\rho] \vdash^{\Theta'}_{\mathcal{G}} \Delta'$.

Proof. By mutual induction on the typing derivation given in each part. The interesting case is the base cases where ρ replaces variable with a value or dually replaces a covariable with a covalue. In both of these cases, we have $x[\rho] = V_s$ or $\alpha[\rho] = E_s$, where the necessary typing derivation for V_s or E_s can be found by induction on the typing derivation of the substitution ρ. □

Corollary B.7 ((Co)Value Substitution Closure)**.** *Let* $(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta) \Rightarrow \rho : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta')$ *be a well-disciplined substitution (*i.e., *possibly using the Untype rule) and* $\Theta \vdash_{\mathcal{G}} A : s$.

a) If $\Gamma \vdash^{\Theta}_{\mathcal{G}} V_s : A ; \Delta$ *is a well-disciplined s-value then so is* $\Gamma' \vdash^{\Theta'}_{\mathcal{G}} V_s[\rho] : A[\rho] ; \Delta'$.
b) If $\Gamma ; E_s : A \vdash^{\Theta}_{\mathcal{G}} \Delta$ *is a well-disciplined s-covalue then so is* $\Gamma' ; E_s[\rho] : A[\rho] \vdash^{\Theta'}_{\mathcal{G}} \Delta'$.

Lemma B.8 (Step Substitution Closure)**.** *Let* $c : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta)$ *be a well-disciplined command and* $(\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta) \Rightarrow \rho : (\Gamma' \vdash^{\Theta'}_{\mathcal{G}} \Delta)'$ *a well-disciplined substitution. If* $c \mapsto c'$ *then* $c[\rho] \mapsto c'[\rho]$.

Proof. By induction on the surrounding heap context surrounding the reduction of c and cases on the possible reduction rules applied. Note that in each reduction rule which assumes that certain sub-(co)terms are (co)values, those (co)values are closed under substitution (Corollary B.7), so the side-condition of the rule still applies. □

In Section 7, we defined the status of a command in terms of the notion of its *needed (co)variables* (written as $\mathrm{NV}(c)$). Importantly, determining a command's status is a decidable judgment with a unique answer for every command.

Property B.9 (Unique status)**.** Every command c is exactly one of (1) finished, (2) stuck, or (3) in progress (*i.e.,* $c \mapsto c'$ for some c').

Theorem 7.5 (Type safety)**.** *a)* Progress*: If* $c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)$*, then* c *is not stuck.*
b) Preservation*: If* $c : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)$ *and* $c \mapsto c'$ *then* $c' : (\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)$.

Proof. Preservation follows from Lemma B.6 by cases on the reduction rule applied and induction on the surrounding heap context. Progress follows by induction on the given

typing derivation of $c = \langle v|A{:}s|e\rangle$, which must end with a *Cut* of the form

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} \ v : A \mid \Delta \quad \Theta \vdash_{\mathcal{G}} A : s \quad \Gamma \mid e : A \vdash^{\Theta}_{\mathcal{G}} \ \Delta}{\langle v|s|e\rangle : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \ \Delta)} \ \mathit{Cut}$$

and depends crucially on cases for the discipline symbol s:

- $s = +$: First consider the term v.
  - If $v = \mu\alpha.c'$, then $c \mapsto_{\beta_\mu}$ for any e.
  - Otherwise $v = W$, and we must next consider both W and the coterm e.
    - If $e = \tilde{\mu}x.c'$, then $c \mapsto_{\beta_{\tilde{\mu}}}$ for any W.
    - If $e = \alpha$, then $\alpha \in \mathrm{NV}(c)$, so c is finished for any W.
    - If $e = \mathsf{O}\,C...V_t...E_r...$, then by inversion on the type A, it must be that either W is a λ-abstraction with a matching branch for O (in which case $c \mapsto_{\beta_q}$) or a variable x (in which case $x \in \mathrm{NV}(c)$ so c is finished).
    - Otherwise $e = \mathcal{K}\{p_i.c'_i \mid .^i.\}$. Similarly by inversion on the type A, it must be that either W is a constructor application matching one of p_i (in which case $c \mapsto_{\beta_p}$) or a variable x (in which case $x \in \mathrm{NV}(c)$ so c is finished).
- $s = \sharp$: First consider the coterm e.
  - If $e = \tilde{\mu}x.c'$ and not a covalue, then we must apply the inductive hypothesis to the sub-derivation of c'.
    - If $c' \mapsto$ then $c \mapsto$ by closure under heap contexts.
    - Otherwise c' is finished with $x \notin \mathrm{NV}(c')$ (because then $\tilde{\mu}x.c'$ would be a covalue by Corollary B.4, contradicting the assumption). c must also finished because $c \not\mapsto$ (since $\tilde{\mu}x.c'$ is not a covalue, preventing β_μ and $\phi^{\sharp}_{\tilde{\mu}}$ reduction) and $\mathrm{NV}(c) = \mathrm{NV}(c')$ is not empty.
  - Otherwise $e = E_\sharp$, and we must next consider both $E_\sharp$ and the term v.
    - If $v = \mu\alpha.c'$ then $c \mapsto_{\beta_\mu}$ for any $E_\sharp$.
    - If $v = x$, then $x \in \mathrm{NV}(c)$, so c is finished for any $E_\sharp$ (even a $\tilde{\mu}$-covalue).
    - If $v = \mathsf{K}\,C...E_r...V_t...$, then by inversion on the type A, it must be that either $E_\sharp$ is a $\mathcal{K}$-abstraction with a matching branch for K (in which case $c \mapsto_{\beta_p}$), a $\tilde{\mu}$-covalue (in which case $c \mapsto_{\phi^{\sharp}_{\tilde{\mu}}}$), or a covariable α (in which case $\alpha \in \mathrm{NV}(c)$ so c is finished).
    - Otherwise $v = \lambda\{q_i.c_i \mid .^i.\}$. By inversion on the type A, it must be that either $E_\sharp$ is a destructor application matching one of q_i (in which case $c \mapsto_{\beta_q}$), a $\tilde{\mu}$-covalue (in which case $c \mapsto_{\phi^{\sharp}_{\tilde{\mu}}}$), or a covariable α (in which case $\alpha \in \mathrm{NV}(c)$ so c is finished).
- $s = -$ or $\flat$: follows dually to the above two cases. □