Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing

Paul Downen
University of Oregon, Eugene, OR, USA
pdownen@cs.uoregon.edu

Zena M. Ariola
University of Oregon, Eugene, OR, USA
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 only adding a mechanism for sharing and the extra *polarity shifts* to connect them, which is enough to compile a Haskell-like functional language with user-defined types.

2012 ACM Subject Classification Theory of computation → Type structures

Keywords and phrases call-by-need, polarity, call-by-push-value, control

Digital Object Identifier 10.4230/LIPIcs.CSL.2018.21

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

1 Introduction

Finding a universal intermediate language suitable for compiling and optimizing both strict and lazy functional programs has been a long-sought holy grail for compiler writers. First there was continuation-passing style (CPS) [19, 2], which hard-codes the evaluation strategy into the program itself. In CPS, all the specifics of evaluation strategy can be understood just by looking at the syntax of the program. Second there were monadic languages [13, 17], 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 adjunctive languages [10, 23, 14], 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 [4] 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 [3]: 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 [12], which is a limited form of jump or *goto* statement, to optimize pure functional programs.

So it seems like the quest for a universal intermediate language is still ongoing. To handle all the issues involving evaluation order in modern functional compilers, the following questions, which have been unanswered so far, should also be addressed:

- (Section 3) How do you extend polarity with sharing (*i.e.,* call-by-need)? For example, how do you model the Glasgow Haskell Compiler (GHC) which mixes both call-by-need for ordinary Haskell programs and call-by-value for *unboxed* [18] machine primitives?
- (Section 4) What does a core language need to serve as a compile target for a general functional programming language with user-defined types? What are the *shifts* you need to convert between all three calling conventions? While encoding data types is routine, what do you need to fully encode *co-data types* [9]?
- (Section 5) How do you compile that general functional language to the core intermediate sub-language? And how do you know that it is robust when effects are added?

This paper answers each of these questions. The formal relationship between our intermediate language and both polarity and call-by-push-value (Appendix A). To test the robustness of this idea, we extend it in several directions in the appendix. We generalize to a dual sequent calculus framework that incorporates more calling conventions (specifically, the dual to call-by-need) and connectives not found in functional languages (Appendices B and C).

2 Polarity, data, and co-data

To begin, let's start with a basic language which is the λ-calculus extended with sums, as expressed by the following types and terms:

$$A, B, C ::= X \mid A \to B \mid A \oplus B$$
$$M, N, P ::= x \mid \lambda x.M \mid M\ N \mid \iota_1 M \mid \iota_2 M \mid \textbf{case}\, M\, \textbf{of}\, \{\iota_1 x.N \mid \iota_2 y.P\}$$

As usual, an abstraction $\lambda x.M$ is a term of a function type $A \to B$ and an injection $\iota_i M$ is a term of a sum type $A \oplus B$. Terms of function and sum types are used via application ($M\ N$) and **case** analysis, respectively. Variables x can be of any type, even an atomic type X.

To make this a programming language, we would need to explain how to run programs (say, closed terms of a sum type) to get results. But what should the calling convention be? We could choose to use call-by-value evaluation, wherein a function application $(\lambda x.M)\ N$ is reduced by first evaluating N and then plugging its value in for x, or call-by-name evaluation, wherein the same application is reduced by immediately substituting N for x without further evaluation. We might think that this choice just impacts efficiency, trading off the cost of evaluating an unneeded argument in call-by-value for the potential cost of re-evaluating the same argument many times in call-by-name. However, the choice of calling convention also impacts the properties of the language, and can affect our ability to reason about programs.

Functions are a co-data type [7], so the extensionality law for functions, known as η, expands function terms into trivial λ-abstractions as follows:

$$(\eta_{\to}) \qquad M : A \to B = \lambda x.M\ x \qquad (x \notin FV(M))$$

But once we allow for any computational effects in the language, this law only makes sense with respect to call-by-name evaluation. For example, suppose that we have a non-terminating term Ω (perhaps caused by general recursion) which never returns a value. Then the $\eta_{\to}$ law stipulates that $\Omega = \lambda x.\Omega\ x$. This equality is fine – it does not change the observable behavior of any program – in call-by-name, but in call-by-value, $(\lambda z.5)\ \Omega$ loops forever and $(\lambda z.5)\ (\lambda x.\Omega\ x)$ returns 5. So the full $\eta_{\to}$ breaks in call-by-value.

In contrast, sums are a data type, so one sensible extensionality law for sums, which corresponds to reasoning by induction on the possible cases of a free variable, is expressed by the following law stating that if x has type $A \oplus B$ then it does no harm to **case** on x first:

$$(\eta_{\oplus}) \qquad M = \mathbf{case}\, x\, \mathbf{of} \{\iota_1 y.M[\iota_1 y/x] \mid \iota_2 z.M[\iota_2 z/x]\} \qquad (x : A \oplus B)$$

Unfortunately, this law only makes sense with respect to call-by-value evaluation once we have effects. For example, consider the instance where M is $\iota_1 x$. In call-by-value, variables stand for *values* which are already evaluated because that is all that they might be substituted for. So in either case, when we plug in something like $\iota_i 5$ for x, we get the result $\iota_1(\iota_i 5)$ after evaluating the right-hand side. But in call-by-name, variables range over all terms which might induce arbitrary computation. If we substitute Ω for x, then the left-hand side results in $\iota_1\Omega$ but the right-hand side forces evaluation of Ω with a **case**, and loops forever.

How can we resolve this conflict, where one language feature "wants" call-by-name evaluation and the other "wants" call-by-value? We just could pick one or the other as the default of the language, to the detriment of either functions or sums. Or instead we could integrate the two to get the best of both worlds, and *polarize* the language so that functions are evaluated according to call-by-name, and sums according to call-by-value. That way, both of them have their best properties in the same language, even when effects come into play. Since functions and sums are already distinguished by types, we can leverage the type system to make the call-by-value and -name distinction for us. That is to say, a type A might classify either a call-by-value term, denoted by A_+, or a call-by-name term, denoted by A_-. Put it all together, we get the following polarized typing rules for our basic λ-calculus:

$$A, B, C ::= A_+ \mid A_- \qquad A_-, B_- ::= X^- \mid A_+ \to B_- \qquad A_+, B_+ ::= X^+ \mid A_+ \oplus B_+$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathit{Var} \qquad \frac{\Gamma, x : A_+ \vdash M : B_-}{\Gamma \vdash \lambda x.M : A_+ \to B_-}\ {\to}I \qquad \frac{\Gamma \vdash M : A_+ \to B_- \quad \Gamma \vdash N : A_+}{\Gamma \vdash M\ N : B_-}\ {\to}E$$

$$\frac{\Gamma \vdash M : A_+}{\Gamma \vdash \iota_1 M : A_+ \oplus B_+}\ \oplus I_1 \qquad \frac{\Gamma \vdash M : B_+}{\Gamma \vdash \iota_2 M : A_+ \oplus B_+}\ \oplus I_2$$

$$\frac{\Gamma \vdash M : A_+ \oplus B_+ \quad \Gamma, x : A_+ \vdash N : C \quad \Gamma, y : B_+ \vdash P : C}{\Gamma \vdash \mathbf{case}\, M\, \mathbf{of} \{\iota_1 x.N \mid \iota_2 y.P\} : C}\ \oplus E$$

Note that, with this polarization, injections are treated as call-by-value, in $\iota_i M$ the term M is evaluated before the tagged value is returned. More interestingly, the function call $M\ N$ has two parts: the argument N is evaluated before the function is called as in call-by-value, but this only happens once the result is demanded as in call-by-name.

But there's a problem, just dividing up the language into two has severely restricted the ways we can compose types and terms. We can no longer inject a function into a sum, because a function is negative but a sum can only contain positive parts. Even more extreme, the identity function $\lambda x.x : A \to A$ no longer makes sense: the input must be a positive type and the output a negative type, and A cannot be both positive and negative at once. To get around this restriction, we need the ability to *shift* polarity between positive and negative. That way, we can still compose types and terms any way we want, just like before, and have the freedom of making the choice between call-by-name or -value instead of having the language impose one everywhere.

If we continue the data and co-data distinction that we had between sums and functions above, there are different ways of arranging the two shifts in the literature, depending on the viewpoint. In Levy's call-by-push-value [10] the shift from positive to negative $\Uparrow$ (therein called F) can be interpreted as a data type, where the sequencing operation is subsumed by the usual notion of a **case** on values of that data type, and the reverse shift $\Downarrow$ (therein called U) can be interpreted as co-data type:[1]

$$A_-, B_- ::= \dots \mid \Uparrow A_+ \qquad \frac{\Gamma \vdash M : A_+}{\Gamma \vdash \mathsf{val}\, M : \Uparrow A_+}\Uparrow I \qquad \frac{\Gamma \vdash M : \Uparrow A_+ \quad \Gamma, x : A_+ \vdash N : C}{\Gamma \vdash \mathbf{case}\, M\, \mathbf{of}\{\mathsf{val}\, x.N\} : C}\Uparrow E$$

$$A_+, B_+ ::= \dots \mid \Downarrow A_- \qquad \frac{\Gamma \vdash M : A_-}{\Gamma \vdash \lambda\mathsf{enter}.M : \Downarrow A_-}\Downarrow I \qquad \frac{\Gamma \vdash M : \Downarrow A_-}{\Gamma \vdash M.\mathsf{enter} : A_-}\Downarrow E$$

$M.\mathsf{enter}$ can be seen as sending the request enter to M, and $\lambda\mathsf{enter}.M$ as waiting for that request. In contrast, Zeilberger's calculus of unity [22] takes the opposite view, where the shift $\uparrow$ from positive to negative is co-data and the opposite shift $\downarrow$ is data:

$$A_-, B_- ::= \dots \mid \uparrow A_+ \qquad \frac{\Gamma \vdash M : A_+}{\Gamma \vdash \lambda\mathsf{eval}.M : \uparrow A_+}\uparrow I \qquad \frac{\Gamma \vdash M : \uparrow A_+}{\Gamma \vdash M.\mathsf{eval} : A_+}\uparrow E$$

$$A_+, B_+ ::= \dots \mid \downarrow A_- \qquad \frac{\Gamma \vdash M : A_-}{\Gamma \vdash \mathsf{box}\, M : \downarrow A_-}\downarrow I \qquad \frac{\Gamma \vdash M : \downarrow A_- \quad \Gamma, x : A_- \vdash N : C}{\Gamma \vdash \mathbf{case}\, M\, \mathbf{of}\{\mathsf{box}\, x.N\} : C}\downarrow E$$

Here, we do not favor one form over the other and allow both forms to coexist. In turns out that with only call-by-value and -name evaluation, the two pairs of shifts amount to the same thing (more formally, we will see in Section 5 that they are *isomorphic*). But we will see next in Section 3 how extending this basic language calls both styles of shifts into play.

With the polarity shifts between positive and negative types, we can express every program that we could have in the original unpolarized language. The difference is that now since *both* call-by-value and -name evaluation is denoted by different types, the types themselves signify the calling convention. For call-by-name, this encoding is:

$$[\![X]\!]^- = X^- \qquad [\![A \to B]\!]^- = (\downarrow[\![A]\!]^-) \to [\![B]\!]^- \qquad [\![A \oplus B]\!]^- = \Uparrow((\downarrow[\![A]\!]^-) \oplus (\downarrow[\![B]\!]^-))$$

$$[\![x]\!]^- = x$$

$$[\![M\ N]\!]^- = [\![M]\!]^-(\mathsf{box}\, [\![N]\!]^-) \qquad [\![\lambda x.M]\!]^- = \lambda y.\, \mathbf{case}\, y\, \mathbf{of}\{\mathsf{box}\, x.[\![M]\!]^-\}$$

$$[\![\iota_i M]\!]^- = \mathsf{val}(\iota_i(\mathsf{box}\, [\![M]\!]^-)) \qquad [\![\mathbf{case}\, M\, \mathbf{of}\{\iota_i x_i.N_i\}]\!]^- = \mathbf{case}\, [\![M]\!]^-\, \mathbf{of}\{\mathsf{val}(\iota_i(\mathsf{box}\, x_i)).[\![N_i]\!]^-\}$$

[1] Note that this $\Uparrow E$ rule is an *extension* of the elimination rule for F in call-by-push-value [10], which restricts C to be only a negative type. The impact is that, unlike call-by-push-value, this language allows for *non-value terms* of positive types, similar to SML. The extension is *conservative*, because the interpretation of A_+ values is identical to call-by-push-value, whereas the interpretation of a non-value term of type A_+ would be shifted in call-by-push-value as the computation type $\Uparrow A_+$. This interpretation also illustrates how to compile the extended calculus to the lower-level call-by-push-value by $\Uparrow$-shifting following the standard encoding of call-by-value, where positive non-value terms have an explicit val wherever they may return a value. More details can be found in Appendix A.

where the nested pattern $\mathsf{val}(\iota_i(\mathsf{box}\,x_i))$ is expanded in the obvious way. It converts every type into a negative one, and amounts to boxing up the arguments of injections and function calls. The call-by-value encoding is:

$$
\begin{gathered}
[\![X]\!]^+ = X^+ \qquad [\![A \to B]\!]^+ = \Downarrow([\![A]\!]^+ \to (\Uparrow[\![B]\!]^+)) \qquad [\![A \oplus B]\!]^+ = [\![A]\!]^+ \oplus [\![B]\!]^+ \\
[\![x]\!]^+ = x \\
[\![M\ N]\!]^+ = (([\![M]\!]^+.\mathsf{enter})\ [\![N]\!]^+).\mathsf{eval} \qquad [\![\lambda x.M]\!]^+ = \lambda\mathsf{enter}.\lambda x.\lambda\mathsf{eval}.[\![M]\!]^+ \\
[\![\iota_i M]\!]^+ = \iota_i[\![M]\!]^+ \qquad [\![\mathbf{case}\,M\,\mathbf{of}\{\iota_i x_i.N_i\}]\!]^+ = \mathbf{case}\,[\![M]\!]^+\,\mathbf{of}\{\iota_i x_i.[\![N_i]\!]^+\}
\end{gathered}
$$

It converts every type into a positive one. As such, sum types do not have to change (because, like SML, we have not restricted positive types to only classifying values as in [14]). Instead, the shifts appear in function types: to call a function, we must first enter the abstraction, perform the call, then evaluate the result.

At a basic level, these two encodings 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 encodings are *sound* with respect to equality of terms: if two terms are equal, then their encodings are also equal. We have not yet formally defined equality, so we will return to this question later in Section 5.1.

3 Polarity and sharing

So far we have considered only call-by-value and -name calculi. What about call-by-need, which models sharing and memoization for lazy computation; what would it take to add that, too? The shifts we have are no longer enough: to complete the picture we also require shifts between call-by-need and the other polarities. We need to be able to shift into and out of the positive polarity in order for call-by-need to access data like the sum type. And we also need to be able to shift into and out of the negative polarity for call-by-need to be able to access co-data like the function type. That is a total of four more shifts to connect the ordinary polarized language to the call-by-need world. The question is, how do we align the four different shifts that we saw previously? Since call-by-need only needs access to the positive world for representing data types, we use the data forms of shifts between those two. Dually, since call-by-need only needs access to the negative world for representing co-data types, we use the co-data forms of shifts between those two. We will also need a mechanism for representing sharing. The traditional representation [4] is with **let**-bindings, and so we will do the same. In all, we have:

$$
\begin{array}{ll}
A, B, C ::= A_+ \mid A_- \mid A_\star & A_-, B_- ::= X^- \mid A_+ \to B_- \mid \Uparrow A_+ \mid \uparrow A_+ \mid \uparrow_\star A_\star \\
A_\star, B_\star ::= X^\star \mid {}_\star\!\Uparrow A_+ \mid {}_\star\!\Downarrow A_- & A_+, B_+ ::= X^+ \mid A_+ \oplus B_+ \mid \Downarrow A_- \mid \downarrow A_- \mid \downarrow_\star A_\star
\end{array}
$$

$$
\frac{\Gamma \vdash M : A_\star}{\Gamma \vdash \lambda \mathsf{eval}_\star . M : \uparrow_\star A_\star}\ \uparrow I
\qquad
\frac{\Gamma \vdash M : \uparrow_\star A_\star}{\Gamma \vdash M.\mathsf{eval}_\star : A_\star}\ \uparrow E
$$

$$
\frac{\Gamma \vdash M : A_\star}{\Gamma \vdash \mathsf{box}_\star\, M : \downarrow_\star A_\star}\ \downarrow I
\qquad
\frac{\Gamma \vdash M : \downarrow_\star A_\star \quad \Gamma, x : A_\star \vdash N : C}{\Gamma \vdash \mathbf{case}\, M\, \mathbf{of} \{\mathsf{box}_\star\, x.N\} : C}\ \downarrow E
$$

$$
\frac{\Gamma \vdash M : A_+}{\Gamma \vdash \mathsf{val}_\star\, M : {}_\star\!\Uparrow A_+}\ \Uparrow I
\qquad
\frac{\Gamma \vdash M : {}_\star\!\Uparrow A \quad \Gamma, x : A_+ \vdash N : C}{\Gamma \vdash \mathbf{case}\, M\, \mathbf{of} \{\mathsf{val}_\star\, x.N\} : C}\ \Uparrow E
$$

$$
\frac{\Gamma \vdash M : A_-}{\Gamma \vdash \lambda \mathsf{enter}_\star . M : {}_\star\!\Downarrow A_-}\ \Downarrow I
\qquad
\frac{\Gamma \vdash M : {}_\star\!\Downarrow A_-}{\Gamma \vdash M.\mathsf{enter}_\star : A_-}\ \Downarrow E
$$

$$
\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : C}{\Gamma \vdash \mathbf{let}\, x = M\, \mathbf{in}\, N : C}\ \mathit{Let}
$$

Now, how can a call-by-need λ-calculus with functions and sums be encoded into this polarized setting? We effectively combine both the call-by-name and -value encodings, where a shift is used for call-by-need whenever one is used for either of the other two.

$$
\llbracket X \rrbracket^\star = X^\star \qquad \llbracket A \to B \rrbracket^\star = {}_\star\!\Downarrow((\downarrow_\star \llbracket A \rrbracket^\star) \to (\uparrow_\star \llbracket B \rrbracket^\star)) \qquad \llbracket A \oplus B \rrbracket^\star = {}_\star\!\Uparrow((\downarrow_\star \llbracket A \rrbracket^\star) \oplus (\downarrow_\star \llbracket B \rrbracket^\star))
$$

$$
\begin{aligned}
\llbracket x \rrbracket^\star &= x \\
\llbracket M\ N \rrbracket^\star &= ((\llbracket M \rrbracket^\star . \mathsf{enter}_\star)\ (\mathsf{box}_\star\, \llbracket N \rrbracket^\star)).\, \mathsf{eval}_\star \\
\llbracket \lambda x.M \rrbracket^\star &= \lambda \mathsf{enter}_\star . \lambda y.\, \mathbf{case}\, y\, \mathbf{of} \{\mathsf{box}_\star\, x. \lambda \mathsf{eval}_\star . \llbracket M \rrbracket^\star\} \\
\llbracket \iota_i M \rrbracket^\star &= \mathsf{val}_\star(\iota_i(\mathsf{box}_\star\, \llbracket M \rrbracket^\star)) \\
\llbracket \mathbf{case}\, M\, \mathbf{of} \{\iota_i x_i . N_i\} \rrbracket^\star &= \mathbf{case}\, \llbracket M \rrbracket^\star\, \mathbf{of} \{\mathsf{val}_\star(\iota_i(\mathsf{box}_\star\, x_i)).\llbracket N_i \rrbracket^\star\}
\end{aligned}
$$

The key thing to notice here is what is shared and what is not, to ensure that the encoding correctly aligns with call-by-need evaluation. Both the shifts *into* $\star$, the data type ${}_\star\!\Uparrow A_+$ and co-data type ${}_\star\!\Downarrow A_-$, result in terms that can be shared by a **let**. But the shifts *out of* $\star$ are different: the content M of $\mathsf{box}_\star\, M : \downarrow_\star A_\star$ is still shared, like a data structure, but the content M of $\lambda \mathsf{eval}_\star . M : \uparrow_\star A_\star$ is not, like a λ-abstraction. Therefore, the encoding of an injection $\llbracket \iota_i M \rrbracket^\star$ shares the computation of $\llbracket M \rrbracket^\star$ throughout the lifetime of the returned value, as for the argument of a function call:

$$
\llbracket \mathbf{case}\, \iota_i M\, \mathbf{of} \{\iota_i x_i . N_i\} \rrbracket^\star = \mathbf{let}\, x_i = \llbracket M \rrbracket^\star\, \mathbf{in}\, \llbracket N_i \rrbracket^\star \qquad \llbracket (\lambda x.M) N \rrbracket^\star = \mathbf{let}\, x = \llbracket N \rrbracket^\star\, \mathbf{in}\, \llbracket M \rrbracket^\star
$$

Whereas, the encoding of a function $\llbracket \lambda x.M \rrbracket^\star$, being a value, re-computes $\llbracket M \rrbracket^\star$ every time the function is used, which is formalized by the equational theory in Section 4.4.

4 A multi-discipline intermediate language

So far, we have only considered how sharing interacts with polarity in a small language with functions and sums, but programming languages generally have more than just those two types. For example, both SML and Haskell have pairs so we should include those, too, but when do we have enough of a "representative" basis of types that serves as the core kernel language for the general source language? To define our core intermediate language, we will follow the standard practice (as in CPS) of first defining a more general source language, and then identifying the core sub-language that the entire source can be translated into.

The biggest issue is that faithfully encoding types of various disciplines into a core set of primitives is more subtle than it may at first seem. For example, using Haskell's algebraic data type declaration mechanism, we can define both a binary and ternary sum:

$$\begin{aligned} &\mathbf{data}\ \mathsf{Either}\ a\ b\ \mathbf{where} \\ &\quad \mathsf{Left} : a \to \mathsf{Either}\ a\ b \\ &\quad \mathsf{Right} : b \to \mathsf{Either}\ a\ b \end{aligned} \qquad \begin{aligned} &\mathbf{data}\ \mathsf{Either3}\ a\ b\ c\ \mathbf{where} \\ &\quad \mathsf{Choice1} : a \to \mathsf{Either3}\ a\ b\ c \\ &\quad \mathsf{Choice2} : b \to \mathsf{Either3}\ a\ b\ c \\ &\quad \mathsf{Choice3} : c \to \mathsf{Either3}\ a\ b\ c \end{aligned}$$

But $\mathsf{Either}\ a\ (\mathsf{Either}\ b\ c)$ does not faithfully represent $\mathsf{Either3}\ a\ b\ c$ in Haskell, even though it does in SML. The two types are convertible:

$$\begin{aligned} nest(\mathsf{Choice1}\ x) &= \mathsf{Left}\ x \\ nest(\mathsf{Choice2}\ y) &= \mathsf{Right}(\mathsf{Left}\ y) \\ nest(\mathsf{Choice3}\ z) &= \mathsf{Right}(\mathsf{Right}\ z) \end{aligned} \qquad \begin{aligned} unnest(\mathsf{Left}\ x) &= \mathsf{Choice1}\ x \\ unnest(\mathsf{Right}(\mathsf{Left}\ y)) &= \mathsf{Choice2}\ y \\ unnest(\mathsf{Right}(\mathsf{Right}\ z)) &= \mathsf{Choice3}\ z \end{aligned}$$

but they do not describe the same values. $\mathsf{Either}\ a\ (\mathsf{Either}\ b\ c)$ types both the observably distinct terms Ω and $\mathsf{Right}\ \Omega$ – which can be distinguished by pattern matching – but conversion to $\mathsf{Either3}\ a\ b\ c$ collapses them both to Ω. This is not just an issue of needing $n$ary tuples and sums, the same issue arises when pairs and sums are nested with each other.

To ensure that we model a general enough source language, we will consider one that is *extensible* (*i.e.,* allows for user-defined types encompassing many types found in functional languages) and *multi-discipline* (*i.e.,* allows for programs that mix call-by-value, -name, and -need evaluation). These two features interact with one another: user-defined types can combine parts with different calling conventions. But even though users can define many different types, there is still a fixed core set of types, $\mathcal{F}$, capable of representing them all. For example, an extensible and multi-discipline calculus encompasses both the source and target of the three encodings showed previously in Sections 2 and 3. We now look at the full core intermediate language $\mathcal{F}$, and how to translate general source programs into the core $\mathcal{F}$.

4.1 The functional core intermediate language: $\mathcal{F}$

Our language allows for user-defined data and co-data types. A data type introduces a number of constructors for building values of the type, a co-data type introduces a number of *observers* for observing or interacting with values of the type. Figure 1 presents some important examples that define a *core* set of types, $\mathcal{F}$. The calculus instantiated with just the $\mathcal{F}$ types serves as our core intermediate language, as it contains all the needed functionality.

The data and codata declarations for $\oplus$ and $\to$ correspond to the polarized sum and function types from Section 2, with a slight change of notation: we write $X : +$ instead of X^+. The data declaration of $\oplus$ defines its two *constructors* ι_1 and ι_2, and dually the co-data declaration for $\to$ defines its one *observer* call. The terms of the resulting sum type are exactly as they were presented in Section 2. The function type uses a slightly more verbose notation than the λ-calculus for the sake of regularity: instead of $\lambda x.M$ we have $\lambda\{\mathsf{call}\ \boldsymbol{x}.M\}$ and instead of $M\ N$ we have $M.\mathsf{call}\ N$. That is, dual to a **case** matching on the pattern of a data structure, a λ-abstraction matches on the co-pattern of a co-data observation like $\mathsf{call}\ x$. Besides changing notation, the meaning is the same [7].

There are some points to notice about these two declarations. First, disciplines can be mixed within a single declaration, which is used to define the polarized $\to$ function space that accepts a call-by-value (+) input and returns a call-by-name (−) result, but other

Simple (co-)data types

$$\begin{array}{lll}
\mathbf{data}\,(X{:}+) \oplus (Y{:}+) : + \,\mathbf{where} & \mathbf{data}\,(X{:}+) \otimes (Y{:}+) : + \,\mathbf{where} & \mathbf{data}\,0 : + \,\mathbf{where} \\
\quad \iota_1 : (X{:}+ \vdash X \oplus Y) & \quad (\_,\_) : (X{:}+, Y{:}+ \vdash X \otimes Y) & \mathbf{data}\,1 : + \,\mathbf{where}\,() : (\,\vdash 1) \\
\quad \iota_2 : (Y{:}+ \vdash X \oplus Y) & &
\end{array}$$

$$\begin{array}{lll}
\mathbf{codata}\,(X{:}-) \,\&\, (Y{:}-) : - \,\mathbf{where} & \mathbf{codata}\,\top : - \,\mathbf{where} & \mathbf{codata}\,(X{:}+) \to (Y{:}-) : - \,\mathbf{where} \\
\quad \pi_1 : (\,\mid X \,\&\, Y \vdash X{:}-) & & \quad \mathsf{call} : (X{:}+ \mid X \to Y \vdash Y{:}-) \\
\quad \pi_2 : (\,\mid X \,\&\, Y \vdash Y{:}-) & &
\end{array}$$

Quantifier (co-)data types

$$\begin{array}{l}
\mathbf{data}\,\exists_k(X{:}k{\to}+) : + \,\mathbf{where} \\
\quad \mathsf{pack} : (X\ Y{:}+ \vdash^{Y:k} \exists_k X) \\
\mathbf{codata}\,\forall_k(X{:}k{\to}-) : - \,\mathbf{where} \\
\quad \mathsf{spec} : (\,\mid \forall_k X \vdash^{Y:k} X\ Y{:}-)
\end{array}$$

Polarity shift (co-)data types

$$\begin{array}{ll}
\mathbf{data}\,{\downarrow_{\mathcal{S}}}(X{:}\mathcal{S}) : + \,\mathbf{where} & \mathbf{data}\,{}_{\mathcal{S}}{\Uparrow}(X{:}+) : \mathcal{S}\,\mathbf{where} \\
\quad \mathsf{box}_{\mathcal{S}} : (X{:}\mathcal{S} \vdash {\downarrow_{\mathcal{S}}} X) & \quad \mathsf{val}_{\mathcal{S}} : (X{:}+ \vdash {}_{\mathcal{S}}{\Uparrow} X) \\
\mathbf{codata}\,{\uparrow_{\mathcal{S}}}(X{:}\mathcal{S}) : - \,\mathbf{where} & \mathbf{codata}\,{}_{\mathcal{S}}{\Downarrow}(X{:}-) : \mathcal{S}\,\mathbf{where} \\
\quad \mathsf{eval}_{\mathcal{S}} : (\,\mid {\uparrow_{\mathcal{S}}} X \vdash X{:}\mathcal{S}) & \quad \mathsf{enter}_{\mathcal{S}} : (\,\mid {}_{\mathcal{S}}{\Downarrow} X \vdash X{:}-)
\end{array}$$

Figure 1 The $\mathcal{F}$ functional core set of (co-)data declarations.

combinations are also possible. Second, instead of the function type arrow notation to assign a type to the constructors and observers, we use the turnstyle ($\vdash$) of a typing judgement. This avoids the issue that a function type arrow already dictates the disciplines for the argument and result, limiting our freedom of choice.

The rest of the core $\mathcal{F}$ types exercise all the functionality of our declaration mechanism. The nullary version of sums (0) has no constructors and an empty $\mathbf{case}\,M\,\mathbf{of}\{\}$. We have binary and nullary tuples ($\otimes$, 1), which have terms of the form (M, N) and $()$ and are used by $\mathbf{case}\,M\,\mathbf{of}\{(x, y).M\}$ and $\mathbf{case}\,M\,\mathbf{of}\{().M\}$, respectively. We also have binary and nullary products ($\&$, $\top$), with two and zero observers, respectively. The terms of binary products have the form $\lambda\{\pi_1.M|\pi_2.N\}$ and can be observed as $M.\pi_i$, and the nullary product has the term $\lambda\{\}$ which cannot be observed in any way. The shifts are also generalized to operate generically over the choice of call-by-name ($-$), call-by-value ($+$), and call-by-need ($\star$), which we denote by $\mathcal{S}$. The pair of shifts between $+$ (${\downarrow_{\mathcal{S}}}$, ${}_{\mathcal{S}}{\Uparrow}$) and $-$ (${\uparrow_{\mathcal{S}}}$, ${}_{\mathcal{S}}{\Downarrow}$) for each $\mathcal{S}$ has the same form as in Section 3, where we omit the annotation $\mathcal{S}$ when it is clear from the context.

The last piece of functionality is the ability to introduce *locally quantified* types in a constructor or observer. These quantified type variables are listed as a superscript to the turnstyle, and allow user-defined types to perform type abstraction and polymorphism. Two important examples of type abstraction shown in Figure 1 are the universal ($\forall_k$) and existential ($\exists_k$) quantifiers, which apply to a type function $\lambda X{:}k.A$. We will use the shorthand $\forall X{:}k.A$ for $\forall_k(\lambda X{:}k.A)$ and $\exists X{:}k.A$ for $\exists_k(\lambda X{:}k.A)$. The treatment of quantified types is analogous to System F$_\omega$, where types appear in terms as parameters. For example, the term $\lambda\{\mathsf{spec}\,Y{:}k.M\} : \forall Y{:}k.A$ abstracts over the type variable Y in M, and a polymorphic $M : \forall Y{:}k.A$ can be observed via specialization as $M.\mathsf{spec}\,B : A[B/Y]$. Dually, the term $\mathsf{pack}\,B\ M : \exists Y{:}k.A$ hides the type B in the term $M : A[B/Y]$, and an existential $M : \exists Y{:}k.A$ can be unpacked by pattern matching as $\mathbf{case}\,M\,\mathbf{of}\{\mathsf{pack}\ (Y{:}k)\ (x{:}A).N\}$.

4.2 Syntax

The syntax of our extensible and multi-discipline λ-calculus is given in Figure 2. We refer to each of the three kinds of types ($+$, $-$ and $\star$) as a *discipline* which is denoted by the

$$
\begin{array}{l}
A, B, C ::= X \mid \mathsf{F} \mid \lambda \boldsymbol{X}.A \mid A\ B \qquad \boldsymbol{X} ::= X{:}k \qquad k, l ::= \mathcal{S} \mid k \to l \qquad \mathcal{R}, \mathcal{S}, \mathcal{T} ::= + \mid - \mid \star \\
\mathit{decl} ::= \mathbf{data}\, \mathsf{F}(X{:}k)..: \mathcal{S}\, \mathbf{where}\, \mathsf{K} : (A{:}\mathcal{T}.. \vdash^{\boldsymbol{X}..} \mathsf{F}\, X..).. \\
\qquad\quad \mid \mathbf{codata}\, \mathsf{G}(X{:}k)..: \mathcal{S}\, \mathbf{where}\, \mathsf{O} : (A{:}\mathcal{T}.. \mid \mathsf{G}\, X.. \vdash^{\boldsymbol{X}..} B{:}\mathcal{R}).. \\
\quad p ::= \mathsf{K}\, \boldsymbol{X}..\boldsymbol{y}.. \qquad q ::= \mathsf{O}\, \boldsymbol{X}..\boldsymbol{y}.. \qquad \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} ::= x{:}A \\
M, N ::= x \mid \mathbf{let}\, \boldsymbol{x} = M\, \mathbf{in}\, N \mid M.\mathsf{O}\, B..N.. \mid \mathsf{K}\, B..M.. \mid \lambda\{q_i.M_i \overset{i}{..}\} \mid \mathbf{case}\, M\, \mathbf{of}\{p_i.M_i \overset{i}{..}\}
\end{array}
$$

Figure 2 Syntax of a total, pure functional calculus with (co-)data.

meta-variables $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{T}$. A data declaration has the general form

$$
\begin{array}{rl}
\mathbf{data}\, \mathsf{F}(X_1{:}k_1)..(X_n{:}k_n) : \mathcal{S}\, \mathbf{where} & \mathsf{K}_1 : (A_{11} : \mathcal{T}_{11}..A_{1n} : \mathcal{T}_{1n} \vdash \mathsf{F}\, X_1..X_n) \\
& \vdots \\
& \mathsf{K}_m : (A_{m1} : \mathcal{T}_{m1}..A_{mn} : \mathcal{T}_{mn} \vdash \mathsf{F}\, X_1..X_n)
\end{array}
$$

which declares a new type constructor F and value constructors $\mathsf{K}_1 \dots \mathsf{K}_m$. The dual co-data declaration combines the concepts of functions and products, having the general form

$$
\begin{array}{rl}
\mathbf{codata}\, \mathsf{G}(X_1{:}k_1)..(X_n{:}k_n) : \mathcal{S}\, \mathbf{where} & \mathsf{O}_1 : (A_{11} : \mathcal{T}_{11}..A_{1n} : \mathcal{T}_{1n} \mid \mathsf{G}\, X_1..X_n \vdash B_1 : \mathcal{R}_1) \\
& \vdots \\
& \mathsf{O}_m : (A_{m1} : \mathcal{T}_{m1}..A_{mn} : \mathcal{T}_{mn} \mid \mathsf{G}\, X_1..X_n \vdash B_m : \mathcal{R}_m)
\end{array}
$$

Since an observer is dual to a constructor, the signature is flipped around: the signature for O_1 above can be read as "given parameters of types A_{11} to A_{1n}, O_1 can observe a value of type $\mathsf{G}\, X_1..X_n$ to obtain a result of type B_1."[2]

Notice that we can *also* declare types corresponding to purely call-by-value, -name, and -need versions of sums and functions by instantiating $\mathcal{S}$ with $+$, $-$, and $\star$, respectively:

$$
\begin{array}{ll}
\mathbf{data}\, (X{:}\mathcal{S}) \oplus^{\mathcal{S}} (Y{:}\mathcal{S}) : \mathcal{S}\, \mathbf{where} & \qquad \mathbf{codata}\, (X{:}\mathcal{S}) \xrightarrow{\mathcal{S}} (Y{:}\mathcal{S}) : \mathcal{S}\, \mathbf{where} \\
\quad \iota_1^{\mathcal{S}} : (X{:}\mathcal{S} \vdash X \oplus Y) & \qquad\quad \mathsf{call}^{\mathcal{S}} : (X{:}\mathcal{S} \mid X \xrightarrow{\mathcal{S}} Y \vdash Y{:}\mathcal{S}) \\
\quad \iota_2^{\mathcal{S}} : (Y{:}\mathcal{S} \vdash X \oplus Y) &
\end{array}
$$

So the extensible language subsumes *all* the languages shown in Sections 2 and 3.

4.3 Type System

The kind and type system is given in Figure 3. In the style of system F_ω, the kind system is just the simply-typed λ-calculus at the level of types – so type variables, functions, and applications – where each connective is a constant of the kind declared in the global environment $\mathcal{G}$. It also includes the judgement $(\Gamma \vdash^{\Theta}_{\mathcal{F}})\, \mathbf{ctx}$ for checking that a typing context is well-formed, meaning that each variable in Γ is assigned a well-kinded type with respect to the type variables in Θ and global environment $\mathcal{G}$.

The typing judgement for terms is $\Gamma \vdash^{\Theta}_{\mathcal{G}} M : A : \mathcal{S}$, where $\mathcal{G}$ is a list of declarations, $\Theta = X : k..$ assigns kinds to type variables, and $\Gamma = x : A : \mathcal{S}..$ assigns explicitly-kinded types to value variables. The interesting feature of the type system is the use of the two-level

[2] Both of these notions of data and co-data correspond to *finitary* types, since declarations allow for a finite number of constructors or observers for all data and co-data types, respectively. We could just as well generalize declarations with an infinite number of constructors or observers to also capture *infinitary* types at the usual cost of having infinite branching in **case**s and λs. Since this generalization is entirely mechanical and does not enhance the main argument, we leave it out of the presentation.

$$\frac{\Theta, X : k \vdash_{\mathcal{G}} A : l}{\Theta \vdash_{\mathcal{G}} \lambda X{:}k.A : k \to l} \quad \frac{\Theta \vdash_{\mathcal{G}} A : k \to l \quad \Theta \vdash_{\mathcal{G}} B : k}{\Theta \vdash_{\mathcal{G}} A\ B : l} \quad \frac{}{\Theta, X : k \vdash_{\mathcal{G}} X : k} \quad \frac{(\Theta \vdash_{\mathcal{G}} A : \mathcal{T})..}{(x : A : \mathcal{T}.. \vdash^{\Theta}_{\mathcal{G}})\ \mathbf{ctx}}$$

$$\frac{(\Gamma \vdash^{\Theta}_{\mathcal{G}})\ \mathbf{ctx} \quad \Theta \vdash_{\mathcal{G}} A : \mathcal{S}}{\Gamma, x : A : \mathcal{S} \vdash^{\Theta}_{\mathcal{G}} x : A : \mathcal{S}} \quad \frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} M : A : \mathcal{S} \quad \Gamma, x : A : \mathcal{S} \vdash^{\Theta}_{\mathcal{G}} N : C : \mathcal{R}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{let}\, x{:}A = M\, \mathbf{in}\, N : C : \mathcal{R}} \quad \frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} M : A : \mathcal{S} \quad A =_{\beta\eta} B}{\Gamma \vdash^{\Theta}_{\mathcal{G}} M : B : \mathcal{S}}$$

Given $\mathbf{data}\, \mathsf{F}(X{:}k).. : \mathcal{S}\, \mathbf{where}\, \mathsf{K}_i : (A_{ij} : \mathcal{T}_{ij} \overset{j}{..} \vdash^{Y_{ij} : l_{ij} \overset{j}{..}} \mathsf{F}(X..)) \overset{i}{..} \in \mathcal{G}$, we have the rules:

$$\frac{}{\Theta \vdash_{\mathcal{G}} \mathsf{F} : k \to ..\mathcal{S}}$$

$$\frac{(\Gamma \vdash^{\Theta}_{\mathcal{G}})\ \mathbf{ctx} \quad \Theta \vdash_{\mathcal{G}} \mathsf{F}\, C.. : \mathcal{S} \quad (\Theta \vdash_{\mathcal{G}} B_j : l_{ij}) \overset{j}{..} \quad (\Gamma \vdash^{\Theta}_{\mathcal{G}} M_j : A_{ij}[C/X.., B_j/Y_{ij} \overset{j}{..}] : \mathcal{T}_{ij}) \overset{j}{..}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathsf{K}_i\, B_j \overset{j}{..}\ M_j \overset{j}{..} : \mathsf{F}\, C.. : \mathcal{S}}\ \mathsf{F}I_i$$

$$\frac{\Theta \vdash_{\mathcal{G}} C : \mathcal{R} \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} M : \mathsf{F}\, B.. : \mathcal{S} \quad (\Gamma, x_{ij} : A_{ij}[B/X..] : \mathcal{T}_{ij} \overset{j}{..} \vdash^{\Theta, Y_{ij} : l_{ij} \overset{j}{..}}_{\mathcal{G}} N_i : C : \mathcal{R}) \overset{i}{..}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mathbf{case}\, M\, \mathbf{of}\{(\mathsf{K}_i\, Y_{ij}{:}l_{ij} \overset{j}{..}\ x_{ij}{:}A_{ij} \overset{j}{..}).N_i \overset{i}{..}\} : C : \mathcal{R}}\ \mathsf{F}E$$

Given $\mathbf{codata}\, \mathsf{G}(X{:}k).. : \mathcal{S}\, \mathbf{where}\, \mathsf{O}_i : (A_{ij} : \mathcal{T}_{ij} \overset{j}{..} \mid \mathsf{G}(X..) \vdash^{Y_{ij} : l_{ij} \overset{j}{..}} B_i : \mathcal{R}_i) \overset{i}{..} \in \mathcal{G}$, we have the rules:

$$\frac{}{\Theta \vdash_{\mathcal{G}} \mathsf{G} : k \to ..\mathcal{S}}$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} M : \mathsf{G}\, C'.. : \mathcal{S} \quad (\Theta \vdash_{\mathcal{G}} C_j : l_{ij}) \overset{j}{..} \quad (\Gamma \vdash^{\Theta}_{\mathcal{G}} N_j : A_{ij}[C'/X.., C_j/Y_{ij} \overset{j}{..}] : \mathcal{T}_{ij}) \overset{j}{..}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} M.\mathsf{O}_i\, C_j \overset{j}{..}\ N_j \overset{j}{..} : B_i : \mathcal{R}_i}\ \mathsf{G}E_i$$

$$\frac{(\Gamma \vdash^{\Theta}_{\mathcal{G}})\ \mathbf{ctx} \quad \Theta \vdash_{\mathcal{G}} \mathsf{G}\, C.. : \mathcal{S} \quad (\Gamma, x_{ij} : A_{ij}[C/X..] : \mathcal{T}_{ij} \overset{j}{..} \vdash^{\Theta, Y_{ij} : l_{ij} \overset{j}{..}}_{\mathcal{G}} N_i : B_i : \mathcal{R}_i) \overset{i}{..}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \lambda\big\{(\mathsf{O}_i\, Y_{ij}{:}l_{ij} \overset{j}{..}\ x_{ij}{:}A_{ij} \overset{j}{..}).N_i \overset{i}{..}\big\} : \mathsf{G}\, C.. : \mathcal{S}}\ \mathsf{G}I$$

Figure 3 Type system for the pure functional calculus.

judgement $M : A : \mathcal{S}$, which has the intended interpretation that "M is of type A *and* A is of kind $\mathcal{S}$." The purpose of this compound statement is to ensure that the introduction rules do not create ill-kinded types by mistake. This maintains the invariant that if $\Gamma \vdash^{\Theta}_{\mathcal{G}} M : A : \mathcal{S}$ is derivable then so is $(\Gamma \vdash^{\Theta}_{\mathcal{G}})\ \mathbf{ctx}$ and $\Theta \vdash_{\mathcal{G}} A : \mathcal{S}$.

For example, in the $\mathcal{F}$ environment from Figure 1, a type like $A \otimes B$ requires that both A and B are of kind $+$, so the $\otimes$ introduction rule for closed pairs of closed types is:

$$\frac{\vdash_{\mathcal{F}} M : A : + \quad \vdash_{\mathcal{F}} N : A : +}{\vdash_{\mathcal{F}} (M, N) : A \otimes B : +}\ \otimes I$$

The constraint that $A : +$ and $B : +$ in the premises to $\otimes I$ ensures that $A \otimes B$ is indeed a type of $+$. This idea is also extended to variables introduced by pattern matching at a specific type by placing a two-level constraint on the variables. For example, the $\to$ introduction rule for closed function abstractions is:

$$\frac{x : A : + \vdash_{\mathcal{F}} M : B : -}{\vdash_{\mathcal{F}} \lambda\{\mathsf{call}(x{:}A).M\} : A \to B : -}\ {\to}I$$

Notice how when the variable x is added to the environment, it has the type assignment $x : A : +$ because the declared argument type of $\to$ must be some call-by-value type. If the premise of $\to I$ holds, then $A : +$ and $B : -$, so $A \to B$ is a well-formed type of $-$.

Finally, we also need to check that a global environment $\mathcal{G}$ is well-formed, written $\vdash \mathcal{G}$, which amounts to checking that each declaration is in turn like so:

$$\frac{(X : k.., Y : l.. \vdash_{\mathcal{G}} A : \mathcal{T})..}{\mathcal{G} \vdash \mathbf{data}\, \mathsf{F}(X{:}k).. : \mathcal{S}\, \mathbf{where}\, \mathsf{K} : (A : \mathcal{T}.. \vdash^{Y : l..} \mathsf{F}\, X..)..}$$

$$\frac{(X : k.., Y : l.. \vdash_{\mathcal{G}} A : \mathcal{T}).. \quad (X : k.., Y : l.. \vdash_{\mathcal{G}} B : \mathcal{R})..}{\mathcal{G} \vdash \mathbf{codata}\, \mathsf{G}(X{:}k).. : \mathcal{S}\, \mathbf{where}\, \mathsf{O} : (A : \mathcal{T}.. \mid \mathsf{G}\, X.. \vdash^{Y : l..} B : \mathcal{R})..}$$

$$V ::= V_{\mathcal{S}} : A : \mathcal{S} \qquad V_{+} ::= x \mid \mathsf{K}\, B..V.. \mid \lambda\{q_i.M_i \mid \overset{i}{..}\} \qquad V_{-} ::= M \qquad V_{\star} ::= V_{+}$$
$$F ::= \square.\mathsf{O}\, B..V.. \mid \mathbf{case}\,\square\,\mathbf{of}\{p_i.M_i \overset{i}{..}\} \mid \mathbf{let}\, x{:}A{:}{+} = \square\,\mathbf{in}\, M \mid \mathbf{let}\, x{:}A{:}{\star} = \square\,\mathbf{in}\, H[E[x]]$$
$$E ::= \square \mid F[E] \qquad U ::= \mathbf{let}\, x{:}A{:}{\star} = M\,\mathbf{in}\,\square \qquad H ::= \square \mid U[H]$$
$$T ::= \mathbf{let}\, \boldsymbol{x} = M\,\mathbf{in}\,\square \mid \mathbf{case}\, M\,\mathbf{of}\{p_i.\square \mid \overset{i}{..}\}$$

$$
\begin{array}{rc}
(\beta_{let}) & \mathbf{let}\,\boldsymbol{x} = V\,\mathbf{in}\,M \sim M[V/\boldsymbol{x}] \\
(\beta_{\mathsf{O}}) & \lambda\{..|(\mathsf{O}\,\boldsymbol{Y}..\boldsymbol{x}..).M|..\}.\mathsf{O}\,B..\ N.. \sim \mathbf{let}\,\boldsymbol{x} = N..\,\mathbf{in}\,M[B/\boldsymbol{Y}..] \\
(\beta_{\mathsf{K}}) & \mathbf{case}\,\mathsf{K}\,B..N..\,\mathbf{of}\{..|(\mathsf{K}\,\boldsymbol{Y}..\boldsymbol{x}..).M|..\} \sim \mathbf{let}\,\boldsymbol{x} = N..\,\mathbf{in}\,M[B/\boldsymbol{Y}..] \\
(\eta_{let}) & \mathbf{let}\,x{:}A = M\,\mathbf{in}\,x \sim M \\
(\eta_{\mathsf{G}}) & \lambda\{q_i.(x.q_i) \mid \overset{i}{..}\} \sim x \\
(\eta_{\mathsf{F}}) & \mathbf{case}\,M\,\mathbf{of}\{p_i.p_i \mid \overset{i}{..}\} \sim M \\
(\kappa_F) & F[T[M_i \overset{i}{..}]] \sim T[F[M_i]\overset{i}{..}] \\
(\chi^{\mathcal{S}}) & \mathbf{let}\,y{:}B{:}\mathcal{S} = \mathbf{let}\,x{:}A{:}\mathcal{S} = M_1\,\mathbf{in}\,M_2\,\mathbf{in}\,N \sim \mathbf{let}\,x{:}A{:}\mathcal{S} = M_1\,\mathbf{in}\,\mathbf{let}\,y{:}B{:}\mathcal{S} = M_2\,\mathbf{in}\,N
\end{array}
$$

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} M : A : \mathcal{S} \quad M \sim M' \quad \Gamma \vdash^{\Theta}_{\mathcal{G}} M' : A : \mathcal{S}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} M = M' : A : \mathcal{S}}$$

plus compatibility, reflexivity, symmetry, transitivity

Figure 4 Equational theory for the pure functional calculus.

And we say that $\mathcal{G}'$ *extends* $\mathcal{G}$ if it contains all declarations in $\mathcal{G}$.

4.4 Equational Theory

The equational theory, given in Figure 4, equates two terms of the same type that behave the same in any well-typed context.The axioms of equality are given by the relation $\sim$, and the typed equality judgement is $\Gamma \vdash^{\Theta}_{\mathcal{G}} M = N : A : \mathcal{S}$. Because of the multi-discipline nature of terms, the main challenge is deciding when terms are substitutable, which controls when the β_{let} axiom can fire. For example, $\mathbf{let}\,\boldsymbol{x} = M\,\mathbf{in}\,N$ should immediately substitute M without further evaluation if it is a call-by-name binding, but should evaluate M to a value first before substitution if it is call-by-value. And we need the ability to reason about program fragments (*i.e.,* open terms of any type) wherein a variable x acts like a value in call-by-value *only* if it stands for a value, *i.e.,* we can only substitute values and not arbitrary terms for a call-by-value variable. Thus, we link up the static and dynamic semantics of disciplines: each base kind $\mathcal{S}$ is associated with a different set of substitutable terms $V_{\mathcal{S}}$ called *values.* The set of values for $+$ is the most strict (including only variables, λ-abstractions, and constructions $p[\rho]$ built by plugging in values for the holes in a pattern), $-$ is the most relaxed (admitting every term as substitutable), and $\star$ shares the same notion of value as $+$. A true value, then, is a term $V_{\mathcal{S}}$ belonging to a type of kind $\mathcal{S}$, *i.e.,* $V_{\mathcal{S}} : A : \mathcal{S}$. This way, the calling convention is aligned in both the static realm of types are and dynamic realm of evaluation.

The generic β_{let} axiom relies on the fact that the left-hand side of the axiom is well-typed and every type belongs to (at most) one kind; given $\mathbf{let}\,x{:}A = V\,\mathbf{in}\,M$, then it must be that $A : \mathcal{S}$ and V is of the form $V_{\mathcal{S}} : A : \mathcal{S}$ (both in the current environment). So if $x : A \,\&\, B : -$, then every well-typed binding is subject to substitution via β_{let}, but if $x : A \otimes B : +$ then only a value V_{+} in the sense of call-by-value can be substituted. The corresponding extensionality axiom η_{let} eliminates a trivial **let** binding.

The β_{K} and β_{O} axioms match against a constructor K or observer O, respectively, by selecting the matching response within a **case** or λ-abstraction and binding the parameters via a **let**. Special cases of these axioms for a sum injection and function call are:

$$\mathbf{case}\,\iota_i M\,\mathbf{of}\{\iota_1 \boldsymbol{x}_1.N_1 \mid \iota_2 \boldsymbol{x}_2.N_2\} \sim_{\beta_{\iota_i}} \mathbf{let}\,\boldsymbol{x}_i = M\,\mathbf{in}\,N_i$$
$$\lambda\{\mathsf{call}\,\boldsymbol{x}.N\}.\mathsf{call}\,M \sim_{\beta_{\mathsf{call}}} \mathbf{let}\,\boldsymbol{x} = M\,\mathbf{in}\,N$$

The corresponding extensionality axioms η_{G} and η_{F} apply to each co-data type G and data type F to eliminate a trivial λ and **case**, respectively, and again rely on the fact that the left-hand side of the axiom is well-typed to be sensible. The special cases of these axioms for the sum ($\oplus$) and function ($\to$) connectives of $\mathcal{F}$ are:

$$\mathbf{case}\,M\,\mathbf{of}\{\iota_1 x{:}A.\iota_1 x \mid \iota_2 y{:}B.\iota_2 y\} \sim_{\eta_\oplus} M \qquad \lambda\{\mathsf{call}\,y{:}A.(x.\mathsf{call}\,y)\} \sim_{\eta_\to} x$$

The κ_F axiom implements *commutative conversions* which permute a *frame* F of an evaluation context (E) with a *tail* context T, which brings together the frame with the return result of a block-style expression like a **let** or **case**. Frames represent the building blocks of contexts that demand a result from their hole $\square$. The cases for frames are an observation parameterized by values ($\square.\mathsf{O}\,B..V..$), case analysis ($\mathbf{case}\,\square\,\mathbf{of}\{\dots\}$), a call-by-value binding ($\mathbf{let}\,x{:}A{:}{+} = \square\,\mathbf{in}\,M$), or a call-by-need binding which is needed in its body ($\mathbf{let}\,x{:}A{:}\star = \square\,\mathbf{in}\,H[E[x]]$). As per call-by-need evaluation, variable x is *needed* when it appears in the eye of an evaluation context E, in the context of a *heap* H of other call-by-need bindings for different variables. Tail contexts point out where results are returned from block-style expressions, so the body of any **let** ($\mathbf{let}\,\boldsymbol{x} = M\,\mathbf{in}\,\square$) or the branches of any **case** ($\mathbf{case}\,M\,\mathbf{of}\{p.\square..\}$). Since a **case** can have zero or more branches, a tail context can have zero or more holes.

Finally, the $\chi^{\mathcal{S}}$ axiom re-associates nested **let** bindings, so long as the discipline of their bindings match. The restriction to matching disciplines is because not all combinations are actually associative [14]; namely the following two ways of nesting call-by-value and -name **let**s are *not* necessarily the same when M_1 causes an effect:

$$(\mathbf{let}\,y{:}B{:}{-} = (\mathbf{let}\,x{:}A{:}{+} = M_1\,\mathbf{in}\,M_2)\,\mathbf{in}\,N) \neq (\mathbf{let}\,x{:}A{:}{+} = M_1\,\mathbf{in}\,\mathbf{let}\,y{:}B{:}{-} = M_2\,\mathbf{in}\,N)$$

In the above, the right-hand side evaluates M_1 first, but the left-hand side first substitutes $\mathbf{let}\,x{:}A{:}{+} = M_1\,\mathbf{in}\,M_2$ for y, potentially erasing or duplicating the effect of M_1. For example, when M_1 is the infinite loop Ω and N is a constant result z which does not depend on y, then the right-hand side loops forever, but the left-hand side just returns z. But when the disciplines match, re-association is sound. In particular, notice that the χ^- instance of the axiom is derivable from β_{let}, and the χ^+ instance of the axiom is derivable from κ_F. The only truly novel instance of re-association is for call-by-need, which generalizes the special case of κ_F when the outer variable y happens to be needed.

Some of the axioms of this theory may appear to be weak, but nonetheless they let us derive some useful equalities. For example, the λ-calculus' full η law for functions

$$\frac{\Gamma \vdash^{\Theta}_{\mathcal{F}} M : A \to B : - \quad x \notin \Gamma}{\Gamma \vdash^{\Theta}_{\mathcal{F}} \lambda\{\mathsf{call}\,x{:}A.(M.\mathsf{call}\,x)\} = M : A \to B : -}$$

is derivable from $\eta_\to$ and β_{let}. Furthermore, the sum extensionality law from Section 2, and nullary version for the void type 0

$$\Gamma, x : A_1 \oplus A_2 : + \vdash^{\Theta}_{\mathcal{F}} M = \mathbf{case}\,x\,\mathbf{of}\{\iota_i(y_i{:}A_i).M[\iota_i y_i/x]\overset{i}{..}\} : C : \mathcal{R}$$
$$\Gamma, x : 0 : + \vdash^{\Theta}_{\mathcal{F}} M = \mathbf{case}\,x\,\mathbf{of}\{\} : C : \mathcal{R}$$

are derived from the $\eta_\oplus$, η_0, κ_F, and β_{let} axioms. So typed equality of this strongly-normalizing calculus captures "strong sums" (à la [15]). Additionally, the laws of monadic binding [13] (bind-and-return and bind reassociation) and the F functor of call-by-push-value [10] are instances of the generic $\beta\eta\kappa$ laws for the shift data type ${}_{\mathcal{S}}\Uparrow A$:

$$\Gamma \vdash^{\Theta}_{\mathcal{F}} \mathbf{case}\, \mathsf{box}_{\mathcal{S}}\, V\, \mathbf{of}\{\mathsf{box}_{\mathcal{S}}\, \boldsymbol{x}.M\} =_{\beta_{{}_{\mathcal{S}}\Uparrow}\beta_{let}} M[V/\boldsymbol{x}] : C : \mathcal{R}$$
$$\Gamma \vdash^{\Theta}_{\mathcal{F}} \mathbf{case}\, M\, \mathbf{of}\{\mathsf{box}_{\mathcal{S}}(x{:}A).\mathsf{box}_{\mathcal{S}}\, x\} =_{\eta_p} M : {}_{\mathcal{S}}\Uparrow A : \mathcal{S}$$
$$\Gamma \vdash^{\Theta}_{\mathcal{F}} \mathbf{case}\,(\mathbf{case}\, M\, \mathbf{of}\{\mathsf{box}_{\mathcal{S}}\, \boldsymbol{x}.N\})\, \mathbf{of}\{\mathsf{box}_{\mathcal{T}}\, \boldsymbol{y}.N'\} \quad : C : \mathcal{R}$$
$$=_{\kappa_F} \mathbf{case}\, M\, \mathbf{of}\{\mathsf{box}_{\mathcal{S}}\, \boldsymbol{x}.\mathbf{case}\, N\, \mathbf{of}\{\mathsf{box}_{\mathcal{T}}\, \boldsymbol{y}.N'\}\}$$

Note that in the third equality, commuting conversions can reassociate ${}_{\mathcal{S}}\Uparrow A$ and ${}_{\mathcal{T}}\Uparrow B$ bindings for *any* combination of $\mathcal{S}$ and $\mathcal{T}$, including $-$ and $\star$, because a **case** is *always* strict.

Note that, as usual, the equational theory collapses under certain environments and types due to the nullary versions of some connectives: we saw above that with a free variable $x : 0 : +$ all terms are equal, and so too are any two terms of type $\top$ via $\eta_\top$ (the nullary form of product in $\mathcal{F}$). Even still, there are many important cases where the equational theory is coherent. One particular sanity check is that, in the absence of free variables, the two sum injections $\iota_1()$ and $\iota_2()$ are not equal, as inherited from contextual equivalence.

▶ **Theorem 1** (Closed coherence). *For any global environment $\vdash \mathcal{G}$ extending $\mathcal{F}$, the equality $\vdash_{\mathcal{G}} \iota_1() = \iota_2() : 1 \oplus 1 : +$ is not derivable.*

4.5 Adding effects

So far, we have considered only a pure functional calculus. However, one of the features of polarity is its robustness in the face of computational effects, so let's add some. Two particular effects we can add are *general recursion*, in the form of fixed points, and *control* in the form of μ-abstractions from Parigot's $\lambda\mu$-calculus [16]. To do so, we extend the calculus with the following syntax:

$$M, N ::= \dots \mid \nu\boldsymbol{x}.M \mid \mu\boldsymbol{\alpha}.J \qquad J ::= \langle M \| \alpha \rangle \qquad \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma} ::= \alpha{:}A$$

Fixed-point terms $\nu x{:}A.M$ bind x to the result of M inside M itself. Because fixed points must be unrolled before evaluating their underlying term, their type is restricted to $A : -$. Control extends the calculus with *co-variables* $\alpha, \beta, \dots$ that bind to *evaluation contexts* instead of values, letting programs abstract over and manipulate their control flow. The evaluation context bound to a co-variable α of any type A can be invoked (any number of times) with a term $M : A$ via a jump $\langle M \| \alpha \rangle$ that never returns a result, and the co-variable α of type A can be bound with a μ-abstraction $\mu\alpha{:}A.J$.

To go along with the new syntax, we have some additional type checking rules:

$$\frac{\Gamma, x : A : - \vdash^{\Theta}_{\mathcal{G}} M : A : - \mid \Delta}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \nu x{:}A.M : A : - \mid \Delta} \qquad \frac{J : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A : \mathcal{S}, \Delta)}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \mu\alpha{:}A.J : A : \mathcal{S} \mid \Delta} \qquad \frac{\Gamma \vdash^{\Theta}_{\mathcal{G}} M : A : \mathcal{S} \mid \alpha : A : \mathcal{S}, \Delta}{\langle M \| \alpha \rangle : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \alpha : A : \mathcal{S}, \Delta)}$$

The judgements in other typing rules from Figure 3 are all generalized to $\Gamma \vdash^{\Theta}_{\mathcal{G}} M : A : \mathcal{S} \mid \Delta$. There is also a typing judgement for jumps of the form $J : (\Gamma \vdash^{\Theta}_{\mathcal{F}} \Delta)$, where Θ, Γ, and Δ play the same roles; the only difference is that J is not given a type for its result. Unlike terms, jumps never return. As in the $\lambda\mu$-calculus, the environment Δ is placed on the right because co-variables represent alternative return paths. For example, a term

$x : X : -, y : Y : + \vdash_{\mathcal{F}}^{X:-,Y:+} M : Y : - \mid \beta : Y : +$ could return an X via the main path, as in $M = x$, or a Y via β by aborting the main path, as in $M = \mu\alpha{:}X.\langle y \| \beta\rangle$.

And finally, the equational theory is also extended with the following equality axioms:

$$
\begin{array}{llll}
(\nu) & \nu\boldsymbol{x}.M \sim M[\nu\boldsymbol{x}.M/\boldsymbol{x}] & & \\
(\beta_\mu^\alpha) & \langle \mu\boldsymbol{\alpha}.J \| \beta\rangle \sim J[\beta/\boldsymbol{\alpha}] & (\beta_\mu^F) & F[\mu\boldsymbol{\alpha}.J] : B \sim \mu\beta{:}B.J[\langle F \| \beta\rangle / \langle \Box \| \boldsymbol{\alpha}\rangle] \\
(\eta_\mu) & \mu\alpha{:}A.\langle M \| \alpha\rangle \sim M & (\kappa_\mu) & T[\mu\boldsymbol{\alpha}.\langle M_i \| \beta\rangle \overset{i}{..}] \sim \mu\boldsymbol{\alpha}.\langle T[M_i \overset{i}{..}] \| \beta\rangle
\end{array}
$$

The ν axiom unrolls a fixed point by one step. The two β_μ axioms are standard generalizations of the $\lambda\mu$-calculus: β_μ^α substitutes one co-variable for another, and β_μ^F captures a single frame of a μ-abstraction's evaluation context via a *structural substitution* that replaces one context with another. The κ_μ is the commuting conversion that permutes a μ-abstraction with a tail context T.

5 Encoding user-defined (co-)data types into $\mathcal{F}$

Equipped with both the extensible source language and the fixed $\mathcal{F}$ target language, we are now able to give an encoding of user-defined (co-)data types in terms of just the core $\mathcal{F}$ connectives from Figure 1. Intuitively, each data type is converted to an existential $\oplus$-sum-of-$\otimes$-products and each co-data type is converted to a universal &-product-of-functions, both annotated by the necessary shifts in and out of $+$ and $-$, respectively. The encoding is parameterized by a global environment $\mathcal{G}$ so that we know the overall shape of each declared connective. Given that $\mathcal{G}$ contains the following data declaration of F, the encoding of F is:

$$
\begin{array}{l}
\text{Given } \mathbf{data}\,\mathsf{F}(X{:}k).. : \mathcal{S}\,\mathbf{where}\,\mathsf{K}_i : (A_{ij} : \mathcal{T}_{ij} \overset{j}{..} \vdash^{Y_{ij}:l_{ij} \overset{j}{..}} \mathsf{F}(X..)) \overset{i}{..} \in \mathcal{G} \\
[\![\mathsf{F}]\!]_{\mathcal{G}}^{\mathcal{F}} \triangleq \lambda X{:}k...{}_{\mathcal{S}}\Uparrow((\exists Y_{ij}{:}l_{ij}.\overset{j}{..}((\downarrow_{\mathcal{T}_{ij}} A_{ij}) \otimes \overset{j}{..} 1)) \oplus \overset{i}{..} 0)
\end{array}
$$

Dually, given that $\mathcal{G}$ contains the following co-data declaration of G, the encoding of G is:

$$
\begin{array}{l}
\text{Given } \mathbf{codata}\,\mathsf{G}(X{:}k).. : \mathcal{S}\,\mathbf{where}\,\mathsf{O}_i : (A_{ij} : \mathcal{T}_{ij} \overset{j}{..} \mid \mathsf{G}(X..) \vdash^{Y_{ij}:l_{ij} \overset{j}{..}} B_i : \mathcal{R}_i) \overset{i}{..} \in \mathcal{G} \\
[\![\mathsf{G}]\!]_{\mathcal{G}}^{\mathcal{F}} \triangleq \lambda X{:}k...{}_{\mathcal{S}}\Downarrow((\forall Y_{ij}{:}l_{ij}.\overset{j}{..}((\downarrow_{\mathcal{T}_{ij}} A_{ij}) \to \overset{j}{..}(\Uparrow_{\mathcal{R}_i} B_i))) \,\&\, \overset{i}{..} \top)
\end{array}
$$

However, the previous encodings for call-by-name, -value, and -need functions and sums from Sections 2 and 3 are not exactly the same when we take the corresponding declarations of functions and sums from Section 4; the call-by-name and -value encodings are missing some of the shifts used by the generic encoding, and they all elide the terminators (0, 1, and $\top$). Does the difference matter? No, because the encoded types are still *isomorphic*.

▶ **Definition 2** (Type Isomorphism). An isomorphism between two open types of kind k, written $\Theta \vDash_{\mathcal{G}} A \approx B : k$, is defined by induction on k:

- $\Theta \vDash_{\mathcal{G}} A \approx B : k \to l$ when $\Theta, X : k \vDash_{\mathcal{G}} A\ X \approx B\ X : l$, and
- $\Theta \vDash_{\mathcal{G}} A \approx B : \mathcal{S}$ when, for any x and y, there are terms $x : A : \mathcal{S} \vdash_{\mathcal{G}}^{\Theta} N : B : \mathcal{S}$ and $y : B : \mathcal{S} \vdash_{\mathcal{G}}^{\Theta} M : A : \mathcal{S}$ such that $x{:}A{:}\mathcal{S} \vdash_{\mathcal{G}}^{\Theta} (\mathbf{let}\, y{:}B = N\,\mathbf{in}\, M = x) : A : \mathcal{S}$ and $y{:}B{:}\mathcal{S} \vdash_{\mathcal{G}}^{\Theta} (\mathbf{let}\, x{:}A = M\,\mathbf{in}\, N = y) : B : \mathcal{S}$.

Notice that this is an *open* form of isomorphism: in the base case, an isomorphism between types with free variables is witnessed *uniformly* by a single pair of terms. This uniformity in the face of polymorphism is used to make type isomorphism compatible with the $\forall$ and $\exists$ quantifiers. With this notion of type isomorphism, we can formally state how some of the specific shift connectives are redundant. 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.

▶ **Theorem 3.** *The following isomorphisms hold (under $\vDash_{\mathcal{F}}$) for all $\vdash A : +$ and $\vdash B : -$*

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

But clearly the shifts involving $\star$ are not isomorphic, since none of them even share the same kind. Recognizing that sometimes the generic encoding uses unnecessary identity shifts, and given the algebraic properties of polarized types [6], the hand-crafted encodings $[\![A]\!]^{+}$, $[\![A]\!]^{-}$, and $[\![A]\!]^{\star}$ are isomorphic to $[\![A]\!]^{\mathcal{F}}$.

5.1 Correctness of encoding

Type isomorphisms give us a helpful assurance that the encoding of user-defined (co-)data types into $\mathcal{F}$ is actually a faithful one. In every extension of $\mathcal{F}$ with user-defined (co-)data types, all types are isomorphic to their encoding.

▶ **Theorem 4.** *For all $\vdash \mathcal{G}$ extending $\mathcal{F}$ and $\Theta \vdash_{\mathcal{G}} A : k$, $\Theta \vDash_{\mathcal{G}} A \approx [\![A]\!]^{\mathcal{F}}_{\mathcal{G}} : k$.*

Note that this isomorphism is witnessed by terms in the totally pure calculus (without fixed points or μ-abstractions); the encoding works *in spite of* recursion and control, not because of it. Because of the type isomorphism, we can extract a two-way embedding between terms of type A and terms of the encoded type $[\![A]\!]^{\mathcal{F}}_{\mathcal{G}}$ from the witnesses of the type isomorphism. By the properties of isomorphisms, this embedding respects equalities between terms; specifically it is a certain kind of adjunction called an *equational correspondence* [20].

▶ **Theorem 5.** *For all isomorphic types $\Theta \vDash_{\mathcal{G}} A \approx B : \mathcal{S}$, the terms of type A (*i.e.*, $\Gamma \vdash^{\Theta}_{\mathcal{G}} M : A : \mathcal{S} \mid \Delta$) are in equational correspondence with terms of type B (*i.e.*, $\Gamma \vdash^{\Theta}_{\mathcal{G}} N : B : \mathcal{S} \mid \Delta$).*

This means is that, in the context of a larger program, a single sub-term can be encoded into the core $\mathcal{F}$ connectives without the rest of the program being able to tell the difference. This is useful in optimizing compilers for functional languages which change the interface of particular functions to improve performance, without hampering further optimizations.

The possible application of this encoding in a compiler is as an intermediate language: rather than encoding just one sub-term, exhaustively encoding the whole term translates from a source language with user-defined (co-)data types into the core $\mathcal{F}$ connectives. The essence of this translation is seen in the way patterns and co-patterns are transformed; given the same generic (co-)data declarations listed in Figure 3, the encodings of (co-)patterns are:

$$[\![\mathsf{K}_i\, \boldsymbol{Y}..\; \boldsymbol{x}..]\!]^{\mathcal{F}}_{\mathcal{G}} \triangleq \mathsf{val}_{\mathcal{S}}\left(\iota_2^i\,(\iota_1\,(\mathsf{pack}\,\boldsymbol{Y}..\,(\mathsf{box}_{\mathcal{T}}\,\boldsymbol{x}, ..()))))\right)$$
$$[\![\mathsf{O}_i\, \boldsymbol{Y}..\; \boldsymbol{x}..]\!]^{\mathcal{F}}_{\mathcal{G}} \triangleq \mathsf{enter}_{\mathcal{S}}.\pi_2^i.\pi_1.\mathsf{spec}\,\boldsymbol{Y}...\mathsf{call}\,\boldsymbol{x}...\mathsf{eval}_{\mathcal{R}_i}$$

where ι_2^i denotes i applications of the ι_2 constructor, and π_2^i denotes i projections of the π_2 observer. Using this encoding of (co-)patterns, we can encode (co-)pattern-matching as:

$$[\![\mathbf{case}\,M\,\mathbf{of}\{p_i.N_i\overset{i}{..}\}]\!]^{\mathcal{F}}_{\mathcal{G}} \triangleq \mathbf{case}\,[\![M]\!]_{\mathcal{G}}\,\mathbf{of}\{[\![p_i]\!]_{\mathcal{G}}.[\![N_i]\!]_{\mathcal{G}}\overset{i}{..}\} \qquad [\![\lambda\{q_i.M_i\overset{i}{..}\}]\!]^{\mathcal{F}}_{\mathcal{G}} \triangleq \lambda\{[\![q_i]\!]_{\mathcal{G}}.[\![M_i]\!]_{\mathcal{G}}\overset{i}{..}\}$$

as well as data structures and co-data observations:

$$[\![p[B/\boldsymbol{Y}..,M/\boldsymbol{x}..]]\!]^{\mathcal{F}}_{\mathcal{G}} \triangleq [\![p]\!]^{\mathcal{F}}_{\mathcal{G}}[[\![B]\!]^{\mathcal{F}}_{\mathcal{G}}/\boldsymbol{Y}..,[\![M]\!]^{\mathcal{F}}_{\mathcal{G}}/\boldsymbol{x}..]$$
$$[\![M.(q[B/\boldsymbol{Y}..,N/\boldsymbol{x}..])]\!]^{\mathcal{F}}_{\mathcal{G}} \triangleq [\![M]\!]^{\mathcal{F}}_{\mathcal{G}}.([\![q]\!]^{\mathcal{F}}_{\mathcal{G}}[[\![B]\!]^{\mathcal{F}}_{\mathcal{G}}/\boldsymbol{Y}..,[\![N]\!]^{\mathcal{F}}_{\mathcal{G}}/\boldsymbol{x}..])$$

Note that in the above translation, arbitrary *terms* are substituted instead of just *values* as usual. This encoding of terms with user-defined (co-)data types $\mathcal{G}$ into the core $\mathcal{F}$ types is sound with respect to the equational theory (where Γ and Δ are encoded pointwise).

▶ **Theorem 6.** *If the global environment* $\vdash \mathcal{G}$ *extends* $\mathcal{F}$ *and* $\Gamma \vdash^{\Theta}_{\mathcal{G}} M = N : A \mid \Delta$ *then* $[\![\Gamma]\!]^{\mathcal{F}}_{\mathcal{G}} \vdash^{\Theta}_{\mathcal{F}} [\![M]\!]^{\mathcal{F}}_{\mathcal{G}} = [\![N]\!]^{\mathcal{F}}_{\mathcal{G}} : [\![A]\!]^{\mathcal{F}}_{\mathcal{G}} \mid [\![\Delta]\!]^{\mathcal{F}}_{\mathcal{G}}$.

Since the extensible, multi-discipline language is general enough to capture call-by-value, -name, and -need functional languages – or any combination thereof – this encoding establishes a uniform translation from both ML-like and Haskell-like languages into a common core intermediate language: the polarized $\mathcal{F}$.

6 Conclusion

We have showed here how the idea of polarity can be extended with other calling conventions like call-by-need, which opens up its applicability to the implementation of practical functional languages. In particular, we would like to extend GHC's already multi-discipline intermediate language with the core types in $\mathcal{F}$. Since it already has unboxed types [18] 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 eval request or unpacking a val, 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 $\mathsf{Int} \to {\Uparrow_{\star}}\,{}_{\star}{\Downarrow}(\mathsf{Int} \to {\uparrow}\,\mathsf{Int})$ 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 $\forall X{:}\star.\forall Y{:}\star.\forall Z{:}\star.(X \to Y \to Z) \to [X] \to [Y] \to [Z]$ 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: $\forall X{:}\star.\forall Y{:}\star.\forall Z{:}\star.{\Downarrow}({\downarrow}X \to {\downarrow}Y \to {\uparrow}Z) \to {\downarrow}[X] \to {\downarrow}[Y] \to {\uparrow}[Z]$ the interface in the type spells out that the higher-order function uses the faster two-argument calling convention.

References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. `doi:10.1093/logcom/2.3.297`.

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

3 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, jun 2011. Springer Berlin Heidelberg. `doi:10.1007/978-3-642-21691-6_6`.

4 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. `doi:10.1145/199448.199507`.

5 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. `doi:10.1145/351240.351262`.

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

7 Paul Downen and Zena M. Ariola. The duality of construction. In Zhong Shao, editor, *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, apr 2014. `doi:10.1007/978-3-642-54833-8_14`.

8 Paul Downen and Zena M. Ariola. A tutorial on computational classical logic and the sequent calculus. *Journal of Functional Programming*, 28:e3, 2018. `doi:10.1017/S0956796818000023`.

9 Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In David H. Pitt, Axel Poigné, and David E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–157, Berlin, Heidelberg, sep 1987. Springer Berlin Heidelberg. `doi:10.1007/3-540-18508-9_24`.

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

11 Paul Blain Levy. *Jumbo λ-Calculus*, pages 444–455. Springer Berlin Heidelberg, Berlin, Heidelberg, jul 2006. `doi:10.1007/11787006_38`.

12 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, jun 2017. ACM. `doi:10.1145/3062341.3062380`.

13 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. URL: `http://dl.acm.org/citation.cfm?id=77350.77353`.

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

15 Guillaume Munch-Maccagnoni and Gabriel Scherer. Polarised intermediate representation of lambda calculus with sums. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS 2015, pages 127–140. IEEE, jul 2015. `doi:10.1109/LICS.2015.22`.

16 Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning: International Conference*, LPAR '92, pages 190–201, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. `doi:10.1007/BFb0013061`.

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

18 Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In John Hughes, editor, *Functional Programming Languages and Computer Architecture: 5th ACM Conference*, pages 636–666, Berlin, Heidelberg, aug 1991. Springer Berlin Heidelberg. `doi:10.1007/3540543961_30`.

19 Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975. `doi:10.1016/0304-3975(75)90017-1`.

20 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, nov 1993. `doi:10.1007/BF01019462`.

21 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. `doi:10.1145/944705.944723`.

22 Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1):660–96, 2008. doi:10.1016/j.apal.2008.01.001.

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

A Related Work

There have been several polarized languages [10, 23, 14], 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 [1]; 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 – evaluate 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 [8].

Other restrictions vary between different frameworks. First, where computation can happen? In Levy's call-by-push-value (CBPV) [10], value types (corresponding to positive types) only describe values and computation can only occur at computation types (corresponding to negative types), but in Munch-Maccagnoni's system L [14] computation can occur at *any* type. Zeilberger's calculus of unity (CU) [22], 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 both CU and CBPV only deal with *substitutable* entities, to the exclusion of named computations which may not be duplicated or deleted. Second, what types can variables have? In CBPV variables always have positive types, but in CU variables have negative types or positive *atomic* types (and dually co-variables have positive types or negative atomic types). These restrictions explain why the two frameworks chose their favored shifts: $\Uparrow$ introduces a positive variable and $\downarrow$ introduces a negative one, and in the setting of the sequent calculus $\Downarrow$ introduces a negative co-variable and $\uparrow$ introduces a positive one. They also explain CU'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 has no restrictions on the types of (co-)variables.

In both of these ways, the language presented here is spiritually closest to system L. One reason is that call-by-need forces more generality into the system: if there is neither computation nor variables of call-by-need types, then there is no point of sharing work. However, the call-by-value and -name sub-language can still be reduced down to the more restrictive style of CBPV and CU. We showed here that the two styles of positive and negative shifts are isomorphic, so the only difference is reduction to the appropriate normal form. Normalizing the dynamic focusing reductions – originally named ς [21] – along with commuting conversions (κ) and let substitution (β_{let}) is a transformation into a focused term of negative type (where a shift can be applied for positive terms). Negative variables $x{:}A{:}-$ are eliminated by substituting $y.\mathsf{enter}$ for x where $y{:}\Downarrow A{:}+$, and the (co-)variables forbidden in CU can be eliminated by type-directed η-expansion into nested (co-)patterns.

The data and co-data mechanism used here extends the "jumbo" connectives of Levy's jumbo λ-calculus [11] 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 [23] definition of types via (co-)patterns, which is fully dual, extended with sharing.

Simple (co-)data types

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

Quantifier (co-)data types

$$\begin{array}{ll}
\mathbf{data}\,\exists_k(X{:}k {\to} +) : + \,\mathbf{where} & \mathbf{codata}\,\forall_k(X{:}k {\to} -) : - \,\mathbf{where} \\
\quad \mathsf{pack} : (X\ Y{:}+ \vdash^{Y:k} \exists_k X) & \quad \mathsf{spec} : (\mid \forall_k X \vdash^{Y:k} X\ Y{:}-)
\end{array}$$

Polarity shift (co-)data types

$$\begin{array}{ll}
\mathbf{data}\,{\downarrow_{\mathcal{S}}}(X{:}\mathcal{S}) : + \,\mathbf{where} & \mathbf{data}\,{}_{\mathcal{S}}{\Uparrow}(X{:}+) : \mathcal{S} \,\mathbf{where} \\
\quad \mathsf{box}_{\mathcal{S}} : (X{:}\mathcal{S} \vdash {\downarrow_{\mathcal{S}}} X) & \quad \mathsf{val}_{\mathcal{S}} : (X{:}+ \vdash {}_{\mathcal{S}}{\Uparrow} X) \\
\\
\mathbf{codata}\,{\uparrow_{\mathcal{S}}}(X{:}\mathcal{S}) : - \,\mathbf{where} & \mathbf{codata}\,{}_{\mathcal{S}}{\Downarrow}(X{:}-) : \mathcal{S} \,\mathbf{where} \\
\quad \mathsf{eval}_{\mathcal{S}} : (\mid {\uparrow_{\mathcal{S}}} X \vdash X{:}\mathcal{S}) & \quad \mathsf{enter}_{\mathcal{S}} : (\mid {}_{\mathcal{S}}{\Downarrow} X \vdash X{:}-)
\end{array}$$

Figure 5 The $\mathcal{D}$ dual core set of (co-)data declarations.

B A dual multi-discipline sequent calculus

So far, we have seen how the extensible functional calculus enables multi-discipline programming and can represent many user-defined types with mixed disciplines via encodings. The advantage of this calculus is that it's close to an ordinary core calculus for functional programs, but the disadvantage is its incomplete *symmetries*. Most $\mathcal{F}$ types have a dual counterpart (& and $\oplus$, $\forall$ and $\exists$, *etc.*,) but types like $\otimes$ and $\to$ do not. The disciplines $+$ and $-$ represent opposite calling conventions, but the opposite of call-by-need ($\star$) is missing. To complete the picture, we now consider a fully *dual* calculus, which is based on the symmetric setting of the classical sequent calculus.

B.1 The dual core intermediate language: $\mathcal{D}$

In contrast with functional (co-)data declarations, dual calculus allows for symmetric data and co-data type declarations that are properly dual to one another: they can have multiple inputs to the left (of $\vdash$) and multiple outputs to the right (of $\vdash$). This dual notion of (co-)data

$$A, B, C ::= X \mid \mathsf{F} \mid \lambda \boldsymbol{X}.A \mid A\ B \quad \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z} ::= X{:}k \quad k, l ::= \mathcal{S} \mid k \to l \quad \mathcal{R}, \mathcal{S}, \mathcal{T} ::= + \mid - \mid \star \mid \bigstar$$

$$\begin{aligned} decl ::=\ & \mathbf{data}\, \mathsf{F}\, X{:}k.. : \mathcal{S}\, \mathbf{where}\, \mathsf{K} : (A : \mathcal{T}.. \vdash^{\boldsymbol{Y}..} \mathsf{F}\, X.. \mid B : \mathcal{R}..) \\ \mid\ & \mathbf{codata}\, \mathsf{G}\, X{:}k.. : \mathcal{S}\, \mathbf{where}\, \mathsf{O} : (A : \mathcal{T}.. \mid \mathsf{G}\, X.. \vdash^{\boldsymbol{Y}..} B : \mathcal{R}..) \end{aligned}$$

$$c ::= \langle v \| e \rangle$$

$$v ::= x \mid \mu \boldsymbol{\alpha}.c \mid \nu \boldsymbol{x}.v \mid \lambda\{q_i.c_i \mid \overset{i}{..}\} \mid \mathsf{K}\, A..e..v.. \qquad p ::= \mathsf{K}\, \boldsymbol{Y}..\boldsymbol{\alpha}..\boldsymbol{x}.. \qquad \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} ::= x{:}A$$

$$e ::= \alpha \mid \tilde{\mu} \boldsymbol{x}.c \mid \tilde{\nu} \boldsymbol{\alpha}.e \mid \tilde{\lambda}\{p_i.c_i \mid \overset{i}{..}\} \mid \mathsf{O}\, A..v..e.. \qquad q ::= \mathsf{O}\, \boldsymbol{Y}..\boldsymbol{x}..\boldsymbol{\alpha}.. \qquad \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\delta} ::= \alpha{:}A$$

Figure 6 Syntax of the dual calculus.

is strictly more expressive, and lets us declare the new connectives like so:

$$\begin{array}{ll} \mathbf{codata}\, (X{:}-) \parr (Y{:}-) : -\, \mathbf{where} & \mathbf{codata}\, \bot : -\, \mathbf{where} \\ \quad [\_, \_] : (\ \mid X \parr Y \vdash X : -, Y : -) & \quad [] : (\ \mid \bot \vdash\ ) \\ \mathbf{data}\, \ominus(X{:}-) : +\, \mathbf{where} & \mathbf{codata}\, \neg(X{:}+) : -\, \mathbf{where} \\ \quad \mathsf{cont} : (\ \vdash \ominus X \mid X : -) & \quad \mathsf{throw} : (X : + \mid \neg X \vdash\ ) \end{array}$$

Note how these types rely on the newfound flexibility of having zero outputs (for $\bot$ and $\neg$) and more than one output (for $\parr$ and $\ominus$). These four types generalize $\mathcal{F}$, and decompose function types into the more primitive negative disjunction and negation types, analogous to the encoding of functions in classical logic: $A \to B \approx (\neg A) \parr B$. The full set of dual core $\mathcal{D}$ connectives is given in Figure 5.

B.2 Syntax

The syntax of the dual calculus is given in Figure 6 which is split in two: dual to *terms* (v) which give an answer are *co-terms* (e) which ask a question. Each of the features from the functional language are divided into one of two camps. Variables x, μ-abstractions $\mu\boldsymbol{\alpha}.c$, fixed points $\nu\boldsymbol{x}.v$, objects of co-data types $\lambda\{\ldots\}$, and data structures like $\iota_i v$ are all terms. Dually, co-variables α, $\tilde{\mu}$-abstractions $\tilde{\mu}\boldsymbol{x}.c$ (analogous to **let** and dual to μ), co-fixed points $\tilde{\nu}\boldsymbol{\alpha}.e$, case analysis of data structures $\tilde{\lambda}\{\ldots\}$ (dual to co-data objects) and co-data observations like $\pi_i e$ (dual to data structures) are all co-terms. A command c is analogous to a jump, and puts together an answer (v) with a question (e). The dual calculus can be seen as inverting elimination forms to the other side of a jump $\langle M \| \alpha \rangle$, expanding the role of α. By giving a body to observations themselves, co-patterns q introduce names for *all* sub-components of observations dual to patterns p: for example, the co-pattern of a projection $\pi_i[\alpha{:}A_i] : A_1 \,\&\, A_2$ is perfectly symmetric to the pattern of an injection $\iota_i(x{:}A_i) : A_1 \oplus A_2$.

In types, there is a dual set of disciplines and connectives. The base kind $\star$ signifies the dual to call-by-need ($\bigstar$); it shares delayed questions the same way call-by-need shares delayed answers. The negative co-data type constructors $\parr$ and $\bot$ of $\mathcal{D}$ are dual to the positive connectives $\otimes$ and 1, respectively: they introduce a co-pair $[e, e'] : A \parr B$, which is a pair of co-terms $e : A$ and $e' : B$ accepting inputs of type A and B, and the co-unit $[] : \bot$. Objects of co-data types respond to observations by inverting their *entire* structure and then running a command. For $\&$ this looks like $\lambda\{\pi_1[\alpha{:}A].c_1 \mid \pi_2[\beta{:}B].c_2\} : A \,\&\, B$ and for $\parr$ like $\lambda\{[x{:}A, \beta{:}B].c\} : A \parr B$. In lieu of a non-symmetric function type, we instead have two dual negations: the data type constructor $\ominus : - \to +$ and the co-data type constructor $\neg : + \to -$ which introduce the (co-)patterns $\mathsf{cont}(\alpha{:}A) : \ominus A$ and $\mathsf{throw}[x{:}A] : \neg A$. These particular

$$\frac{\Theta, X:k \vdash_{\mathcal{G}} A:l}{\Theta \vdash_{\mathcal{G}} \lambda X{:}k.A : k \to l} \quad \frac{\Theta \vdash_{\mathcal{G}} A : k \to l \quad \Theta \vdash_{\mathcal{G}} B : k}{\Theta \vdash_{\mathcal{G}} A\ B : l} \quad \frac{}{\Theta, X{:}k \vdash_{\mathcal{G}} X : k} \quad \frac{(\Theta \vdash_{\mathcal{G}} A : \mathcal{T})..\quad (\Theta \vdash_{\mathcal{G}} B : \mathcal{R})..}{(x : A.. \vdash^{\Theta}_{\mathcal{G}} \beta : B..)\,\mathbf{ctx}}$$

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

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

$$\frac{\Gamma, x : A \vdash^{\Theta}_{\mathcal{D}} v : A \mid \Delta \quad \Theta \vdash_{\mathcal{D}} A : -}{\Gamma \vdash^{\Theta}_{\mathcal{D}} \nu x{:}A.v : A \mid \Delta}RR \qquad \frac{\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{D}} \alpha : A, \Delta \quad \Theta \vdash_{\mathcal{D}} A : +}{\Gamma \mid \tilde{\nu}\alpha{:}A.e : A \vdash^{\Theta}_{\mathcal{D}} \Delta}RL$$

$$\frac{\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{D}} \Delta \quad \Theta \vdash_{\mathcal{D}} A =_{\beta\eta} B : \mathcal{S}}{\Gamma \mid e : B \vdash^{\Theta}_{\mathcal{D}} \Delta}TCR \qquad \frac{\Gamma \vdash^{\Theta}_{\mathcal{D}} v : A \mid \Delta \quad \Theta \vdash_{\mathcal{D}} A =_{\beta\eta} B : \mathcal{S}}{\Gamma \vdash^{\Theta}_{\mathcal{D}} v : B \mid \Delta}TCL$$

Given $\mathbf{data}\,\mathsf{F}(X{:}k)..:\mathcal{S}\,\mathbf{where}\,\mathsf{K}_i : (A_{ij} : \mathcal{T}_{ij}\overset{j}{..} \vdash^{Y_{ij}:l_{ij}\overset{j}{..}} \mathsf{F}(X..) \mid B_{ij} : \mathcal{R}_{ij}\overset{j}{..})\overset{i}{..} \in \mathcal{G}$, we have the rules:

$$\frac{}{\Theta \vdash_{\mathcal{G}} \mathsf{F} : k.. \to \mathcal{S}}$$

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

$$\frac{c_i : (\Gamma, x_{ij} : A_{ij}[C/X..]\overset{j}{..} \vdash^{\Theta, Y_{ij}:l_{ij}\overset{j}{..}}_{\mathcal{G}} \alpha_{ij} : B_{ij}[C/X..]\overset{j}{..}, \Delta)\overset{i}{..}}{\Gamma \mid \tilde{\lambda}\big\{(\mathsf{K}_i\, Y_{ij}{:}l_{ij}\overset{j}{..}\ x_{ij}{:}A_{ij}\overset{j}{..}\ x_{ij}{:}A_{ij}\overset{j}{..}).c_i\overset{i}{..}\big\} : \mathsf{F}\,C.. \vdash^{\Theta}_{\mathcal{G}} \Delta}\ \mathsf{F}L$$

Given $\mathbf{codata}\,\mathsf{G}(X{:}k)..:\mathcal{S}\,\mathbf{where}\,\mathsf{O}_i : (A_{ij} : \mathcal{T}_{ij}\overset{j}{..} \mid \mathsf{G}(X..) \vdash^{Y_{ij}:l_{ij}\overset{j}{..}} B_{ij} : \mathcal{R}_{ij}\overset{j}{..})\overset{i}{..} \in \mathcal{G}$, we have the rules:

$$\frac{}{\Theta \vdash_{\mathcal{G}} \mathsf{G} : k.. \to \mathcal{S}}$$

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

$$\frac{c_i : (\Gamma, x_{ij} : A_{ij}[C/X..]\overset{j}{..} \vdash^{\Theta, Y_{ij}:l_{ij}\overset{j}{..}}_{\mathcal{G}} \alpha_{ij} : B_{ij}[C/X..]\overset{j}{..}, \Delta)\overset{i}{..}}{\Gamma \vdash^{\Theta}_{\mathcal{G}} \lambda\big\{[\mathsf{O}_i\, Y_{ij}{:}l_{ij}\overset{j}{..}\ x_{ij}{:}A_{ij}\overset{j}{..}\ \alpha_{ij}{:}B_{ij}\overset{j}{..}].c_i\overset{i}{..}\big\} : \mathsf{G}\,C.. \mid \Delta}\ \mathsf{G}R$$

Figure 7 Type system for the dual calculus.

forms of negation are chosen because they are *involutive* up to isomorphism (as defined next in Appendix C); their two compositions are identities on types: $\ominus(\neg A) \approx A$ and $\neg(\ominus B) \approx B$ for any $A : +$ and $B : -$. Function types can be faithfully represented as $A \to B \approx (\neg A) \mathbin{⅋} B$.

B.3 Type system

The type system of $\mathcal{D}$ is given in Figure 7. One change from the functional calculus' type system is the use of the single-level typing judgement $v : A$ instead of the two-level $M : A : \mathcal{S}$. This is possible because of the sequent calculus' *sub-formula property* – *Cut* is the only inference rule that introduces arbitrary new types in the premises. By just checking that the type of a *Cut* makes sense in the current environment, well-formedness can be separated from typing: if the conclusion of a derivation is well-formed (*i.e.,* $(\Gamma \vdash^{\Theta}_{\mathcal{D}} \Delta)\,\mathbf{ctx}$), then every judgement in the derivation is too. There is also a typing judgement for co-terms; $\Gamma \mid e : A \vdash^{\Theta}_{\mathcal{D}} \Delta$ means that e works with a term of type A in the environments Θ, Γ, Δ.

$$
\begin{array}{llll}
V_{+} ::= x \mid \mathsf{K}\, B..E..V.. \mid \lambda\{q.c..\} & V_{\varstar} ::= V_{+} \mid \mu\boldsymbol{\alpha}.H[\langle V_{\varstar} \| \alpha\rangle] & V_{-} ::= v & V_{\star} ::= V_{+} \\
E_{-} ::= \alpha \mid \mathsf{O}\, B..V..E.. \mid \tilde{\lambda}\{p.c..\} & E_{\star} ::= E_{-} \mid \tilde{\mu}\boldsymbol{x}.H[\langle x \| E_{\star}\rangle] & E_{+} ::= e & E_{\varstar} ::= E_{-} \\
H ::= \Box \mid \langle v \| \tilde{\mu}x{:}A{:}\star.H\rangle \mid \langle \mu\alpha{:}A{:}\varstar.H \| e\rangle & & &
\end{array}
$$

$$
\begin{array}{llllll}
(\beta_{\mu}) & \langle \mu\boldsymbol{\alpha}.c \| E\rangle \sim c[E/\boldsymbol{\alpha}] & (\eta_{\tilde{\mu}}) & \tilde{\mu}x{:}A.\langle x \| e\rangle \sim e & (\nu) & \nu\boldsymbol{x}.v \sim v[\nu\boldsymbol{x}.v/\boldsymbol{x}] \\
(\beta_{\tilde{\mu}}) & \langle V \| \tilde{\mu}\boldsymbol{x}.c\rangle \sim c[V/\boldsymbol{x}] & (\eta_{\mu}) & \mu\alpha{:}A.\langle v \| \alpha\rangle \sim v & (\tilde{\nu}) & \tilde{\nu}\boldsymbol{\alpha}.e \sim e[\nu\boldsymbol{\alpha}.e/\boldsymbol{\alpha}]
\end{array}
$$

$$
\begin{array}{ll}
(\beta_{\mathsf{O}}) & \langle \lambda\{.. \mid [\mathsf{O}\, \boldsymbol{Y}..\boldsymbol{x}..\boldsymbol{\alpha}..].c \mid ..\} \,\|\, \mathsf{O}\, B..v..e..\rangle \sim \langle v.. \| \tilde{\mu}\boldsymbol{x}...\langle \mu\boldsymbol{\alpha}...c[B/Y..] \| e..\rangle\rangle \\
(\beta_{\mathsf{K}}) & \langle \mathsf{O}\, B..e..v.. \| \tilde{\lambda}\{.. \mid (\mathsf{O}\, \boldsymbol{Y}..\boldsymbol{\alpha}..\boldsymbol{x}..).c \mid ..\}\rangle \sim \langle \mu\boldsymbol{\alpha}...\langle v.. \| \tilde{\mu}\boldsymbol{x}...c[B/Y..]\rangle \| e..\rangle
\end{array}
$$

$$
\begin{array}{llll}
(\eta_{\mathsf{G}}) & \lambda\{q_i.\langle x \| q_i\rangle \overset{i}{..}\} \sim x & (\chi^{\star}) & \langle \mu\alpha{:}A{:}\star.\langle v \| \tilde{\mu}y{:}B{:}\star.c\rangle \| e\rangle \sim \langle v \| \tilde{\mu}y{:}B{:}\star.\langle \mu\alpha{:}A{:}\star.c \| e\rangle\rangle \\
(\eta_{\mathsf{F}}) & \tilde{\lambda}\{p_i.\langle p_i \| \alpha\rangle \overset{i}{..}\} \sim \alpha & (\chi^{\varstar}) & \langle v \| \tilde{\mu}y{:}B{:}\varstar.\langle \mu\alpha{:}A{:}\varstar.c \| e\rangle\rangle \sim \langle \mu\alpha{:}A{:}\varstar.\langle v \| \tilde{\mu}y{:}B{:}\varstar.c\rangle \| e\rangle
\end{array}
$$

$$
\frac{c_i : (\Gamma \vdash^{\ominus}_{\mathcal{D}} \Delta) \quad c_1 \sim c_2}{c_1 = c_2 : (\Gamma \vdash^{\ominus}_{\mathcal{D}} \Delta)}
\qquad
\frac{\Gamma \vdash^{\ominus}_{\mathcal{D}} v_i : A \mid \Delta \quad v_1 \sim v_2}{\Gamma \vdash^{\ominus}_{\mathcal{D}} v_1 = v_2 : A \mid \Delta}
\qquad
\frac{\Gamma \vdash^{\ominus}_{\mathcal{D}} e_i : A \mid \Delta \quad e_1 \sim e_2}{\Gamma \mid e_1 = e_2 : A \vdash^{\ominus}_{\mathcal{D}} \Delta}
$$

plus compatibility, reflexivity, symmetry, transitivity

Figure 8 Equational theory for the dual calculus.

B.4 Equational theory

Lastly, we have the equational theory in Figure 8. The dualities of evaluation – between variable and co-variable bindings, data and co-data, values (answers) and evaluation contexts (questions) – are more readily apparent than $\mathcal{F}$. In particular, the notion of substitution discipline for $\mathcal{S}$ is now fully dual as in [7]: a subset of terms (*values* $V_{\mathcal{S}}$) and a subset of co-terms (*co-values* $E_{\mathcal{S}}$) which are substitutable, giving the known dualities between call-by-value $(+)$ and -name $(-)$ [5] and $\star$ and $\varstar$ [3]. The χ axioms reassociate variable and co-variable bindings, and the important cases are for $\star$ (corresponding to $\chi^{\star}$ of **let**s) and $\varstar$. Also note the lack of commuting conversions κ; these follow from the μ axiom.

C Encoding fully dual (co-)data types into $\mathcal{D}$

Now let's looks at the fully dual version of the functional encoding from Section 5. Thanks to the generic notion of shifts, the encoding of dual (co-)data into the core $\mathcal{D}$ connectives is similar to the functional encoding, except that in place of the function type $A \to B$ we use the classical representation $(\ominus A) \parr B$. For the generic (co-)data declarations in Figure 7, we have the following definition:

$$
\begin{aligned}
[\![\mathsf{F}]\!]^{\mathcal{D}}_{\mathcal{G}} &\triangleq \lambda\boldsymbol{X}...{}_{\mathcal{S}}\Uparrow((\exists\boldsymbol{Y}_{ij}\overset{j}{..}.((\ominus(\uparrow_{\mathcal{R}_{ij}} B_{ij})) \otimes \overset{j}{..}((\downarrow_{\mathcal{T}_{ij}} A_{ij}) \otimes \overset{j}{..}1))) \oplus \overset{i}{..}0) \\
[\![\mathsf{G}]\!]^{\mathcal{D}}_{\mathcal{G}} &\triangleq \lambda\boldsymbol{X}...{}_{\mathcal{S}}\Downarrow((\forall\boldsymbol{Y}_{ij}\overset{j}{..}.((\neg(\downarrow_{\mathcal{T}_{ij}} A_{ij})) \parr \overset{j}{..}((\uparrow_{\mathcal{R}_{ij}} B_{ij}) \parr \overset{j}{..}\bot))) \mathbin{\&} \overset{i}{..}\top)
\end{aligned}
$$

The encoding of multi-output data types places a $\ominus$-negates every additional output of a constructor, and the encoding of multi-output co-data is now exactly dual to the data encoding. The encodings of (co-)patterns, (co-)pattern-matching objects, and (co-)data structures follow the above type encoding like so:

$$\begin{aligned}
[\![\mathsf{K}_i\, \boldsymbol{Y}..\ \boldsymbol{\alpha}..\ \boldsymbol{x}..]\!]^{\mathcal{D}}_{\mathcal{G}} &\triangleq \mathsf{val}_{\mathcal{S}}\left(\iota_2^i\,(\iota_1\,(\mathsf{pack}\,\boldsymbol{Y}..\,(\mathsf{cont}[\mathsf{eval}_{\mathcal{R}}\,\boldsymbol{\alpha}],..\,(\mathsf{box}_{\mathcal{T}}\,\boldsymbol{x},..()))))\right)\\
[\![\mathsf{O}_i\, \boldsymbol{Y}..\ \boldsymbol{x}..\ \boldsymbol{\alpha}..]\!]^{\mathcal{D}}_{\mathcal{G}} &\triangleq \mathsf{enter}_{\mathcal{S}}\left[\pi_2^i\,[\pi_1\,[\mathsf{spec}\,\boldsymbol{Y}..\,[\mathsf{throw}[\mathsf{box}_{\mathcal{T}}\,\boldsymbol{x}],..\,[\mathsf{eval}_{\mathcal{R}}\,\boldsymbol{\alpha},..[]]]]]\right]\\
[\![\lambda\{q_i.c_i \overset{i}{..}\}]\!]^{\mathcal{D}}_{\mathcal{G}} &\triangleq \lambda\{[\![q_i]\!]^{\mathcal{D}}_{\mathcal{G}}.[\![c_i]\!]^{\mathcal{D}}_{\mathcal{G}} \overset{i}{..}\}\\
[\![\tilde{\lambda}\{p_i.c_i \overset{i}{..}\}]\!]^{\mathcal{D}}_{\mathcal{G}} &\triangleq \tilde{\lambda}\{[\![p_i]\!]^{\mathcal{D}}_{\mathcal{G}}.[\![c_i]\!]^{\mathcal{D}}_{\mathcal{G}} \overset{i}{..}\}\\
[\![p[C/Y..,e/\alpha..,v/x..]]\!]^{\mathcal{D}}_{\mathcal{G}} &= [\![p]\!]^{\mathcal{D}}_{\mathcal{G}}[[\![C]\!]^{\mathcal{D}}_{\mathcal{G}}/Y..,[\![e]\!]^{\mathcal{D}}_{\mathcal{G}}/\alpha..,[\![v]\!]^{\mathcal{D}}_{\mathcal{G}}/x..]\\
[\![q[C/Y..,v/x..],e/\alpha..]\!]^{\mathcal{D}}_{\mathcal{G}} &= [\![q]\!]^{\mathcal{D}}_{\mathcal{G}}[[\![C]\!]^{\mathcal{D}}_{\mathcal{G}}/Y..,[\![v]\!]^{\mathcal{D}}_{\mathcal{G}}/x..,[\![e]\!]^{\mathcal{D}}_{\mathcal{G}}/\alpha..]
\end{aligned}$$

We also have an analogous notion of type isomorphism. The case for higher kinds is the same, and base isomorphism $\Theta \vDash_{\mathcal{G}} A \approx B : \mathcal{S}$ is witnessed by a pair of inverse commands $c : (x : A \vdash^{\Theta}_{\mathcal{G}} \beta : B)$ and $c' : (y : B \vdash^{\Theta}_{\mathcal{G}} \alpha : A)$ such that both compositions are identities:

$$\langle \mu\beta{:}B.c \| \tilde{\mu}y{:}B.c' \rangle = \langle x \| \alpha \rangle : (x : A \vdash^{\Theta}_{\mathcal{G}} \alpha : A) \quad \langle \mu\alpha{:}A.c' \| \tilde{\mu}x{:}A.c \rangle = \langle y \| \beta \rangle : (y : B \vdash^{\Theta}_{\mathcal{G}} \beta : B)$$

Using type isomorphisms in $\mathcal{D}$, the analogous local and global encodings are sound for fully dual data and co-data types utilizing any combination of $+$, $-$, $\star$, and $\star$ evaluation.

▶ **Theorem 7.** *For all $\vdash \mathcal{G}$ extending $\mathcal{D}$ and $\Theta \vdash_{\mathcal{G}} A : k$, $\Theta \vDash_{\mathcal{G}} A \approx [\![A]\!]^{\mathcal{D}}_{\mathcal{G}} : k$.*

▶ **Theorem 8.** *For all $\vdash \mathcal{G}$ extending $\mathcal{D}$, (co-)terms of type A are in equational correspondence with (co-)terms of type $[\![A]\!]^{\mathcal{D}}_{\mathcal{G}}$, respectively.*

▶ **Theorem 9.** *If $\vdash \mathcal{G}$ extends $\mathcal{D}$ and $c = c' : (\Gamma \vdash^{\Theta}_{\mathcal{G}} \Delta)$ then $[\![c]\!]^{\mathcal{D}}_{\mathcal{G}} = [\![c']\!]^{\mathcal{D}}_{\mathcal{G}} : ([\![\Gamma]\!]^{\mathcal{D}}_{\mathcal{G}} \vdash^{\Theta}_{\mathcal{F}} [\![\Delta]\!]^{\mathcal{D}}_{\mathcal{G}})$.*