Call-By-Unboxed-Value

PAUL DOWNEN, University of Massachusetts Lowell, USA

Call-By-Push-Value has famously subsumed both call-by-name and call-by-value by decomposing programs along the axis of "values" versus "computations." Here, we introduce Call-By-Unboxed-Value which further decomposes programs along an orthogonal axis separating "atomic" versus "complex." As the name suggests, these two dimensions make it possible to express the representations of values as boxed or unboxed, so that functions pass unboxed values as inputs and outputs. More importantly, Call-By-Unboxed-Value allows for an unrestricted mixture of polymorphism and unboxed types, giving a foundation for studying compilation techniques for polymorphism based on *representation irrelevance*. In this regard, we use Call-By-Unboxed-Value to formalize representation polymorphism independently of types; for the first time compiling untyped representation-polymorphic code, while nonetheless preserving types all the way to the machine.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Polymorphism**; • **Theory of computation** → **Program semantics**; **Type theory**.

Additional Key Words and Phrases: Call-by-push-value, focusing, type systems, unboxed types, polymorphism

ACM Reference Format:
Paul Downen. 2024. Call-By-Unboxed-Value. *Proc. ACM Program. Lang.* X, ICFP, Article Y (September 2024), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

High-level polymorphism and low-level machine representations can be like oil and water. The most common implementation techniques avoid mixing them altogether, either specializing all polymorphic code at compile-time (*i.e., monomorphization*) or forcing everything to look the same (*i.e., uniform representation*). Both options have a cost: monomorphization can limit the expressiveness of polymorphism and cause code duplication, while uniform representation can introduce severely costly indirection due to *boxing* that replaces complex data with a pointer.

But there is a third option [15, 19, 47] that attempts to combine the best of both approaches by instead using *representation irrelevance* to compile programs. The main idea is to still allow for polymorphic source code to generalize over different types of data that might be implemented with representations at run-time, but *only* if the choice of representation has no real run-time impact on the generated code. This technique relies on using a static type system to both statically track the representation of each type of value, as well as to reject instances of polymorphism where the compiled machine code would change for different specializations.

One of the biggest complications with implementing representation irrelevance is that the type system—and thus the dividing line between permitted and rejected programs—seems to depend on some ambient notion of "the compiler." For example, consider the polymorphic application function:

$$app :: (a \to b) \to a \to b \qquad\qquad app\ f\ x = f\ x$$

Author's Contact Information: Paul Downen, Paul_Downen@uml.edu, University of Massachusetts Lowell, Lowell, Massachusetts, USA.

ACM 2475-1421/2024/9-ARTY
https://doi.org/XXXXXXX.XXXXXXX

The question is: can a and b have any representation, or must they be statically fixed to some choice (*e.g.*, a pointer) at compile-time? f is a function that is surely represented as a pointer (to a closure), but x has the generic type a. Thus, *app*'s code needs to statically fix a's representation compile-time to find where x is passed in. On the right-hand-side, we have an application $f\ x$ which will return a value of type b, so *app* needs to fix b's representation as well to find where $f\ x$ returns its result.

But wait! We might know the compiler is always going to optimize tail calls so that the final application $f\ x$ will overwrite and reuse *app*$'$*s* stack space. If so, then $f\ x$ doesn't actually return anything to *app* itself—it can't—but instead returns directly to *app*'s original caller. In other words, *app*'s return type b can have any representation *sometimes*, depending on whether or not our compiler will optimize the tail call. The question of when representation is really irrelevant becomes even more murky when we consider other, seemingly minor, variants of *app*:

$$app' :: (a \to b) \to a \to b \qquad\qquad app'\ f = f$$

app$'$ seems to be fine with *any* a and b since all $a \to b$ values are represented as closures, making the choice irrelevant for moving f around. In other words, *app*$'$ can have a *more* generic type than *app*, even though they differ only by a routine η-reduction. There is much left unsaid in this code.

This paper introduces a new parameter-passing paradigm, *Call-By-Unboxed-Value*, where programs fully spell the details needed to unequivocally answer these sorts of questions. Instead of relying on the intuition of seasoned compiler writers to decide when representation is relevant, Call-By-Unboxed-Value provides a single, compiler-independent language with the motto,

> *If you can write it, you can run it.*

In particular, Call-By-Unboxed-Value provides a stable basis for exploring the field of representation irrelevance and polymorphism with the following benefits compared to previous work:

- It provides an unambiguous syntax for separating *complex* versus *atomic* unboxed values, making it possible to predict when atomic values (ultimately stored in registers) will be moved or copied or when the contents of references (ultimately stored in long-term memory) will be read/written, without information about the compiler.
- *All* Call-By-Unboxed-Value programs can be directly compiled and run, as-is, without type checking, to the benefit of compilers with untyped intermediate languages. Instead of type checking, the program is annotated with just enough information about representation that, in addition to the boxed versus unboxed status, spells out where atomic values are held.
- Nevertheless, compilation of Call-By-Unboxed-Value preserves types if it happens to be given a well-typed program, to the benefit of compilers that work with typed intermediate languages. This is in stark contrast with previous work [15, 19], that compiles well-typed source code into impossible-to-type target code.

Happily, Call-By-Unboxed-Value also expresses the efficient higher-order calling conventions [15, 17], where function calls can pass several arguments at once to unknown functions without checking any run-time information. For example, consider the common *zipWith* function:

$$zipWith\ f\ (x{:}xs)\ (y{:}ys) = f\ x\ y : zipWith\ f\ xs\ ys \qquad\qquad zipWith\ f\ xs\ ys = []$$

Ideally, the call $f\ x\ y$ could be compiled as a fast call by just passing x and y in two registers, unpacking f's closure, and jumping to f's code. But this calling convention would *crash* if f is bound to a function expecting three arguments, like $\lambda x\ y\ z.\ (x+y)*z$, or to a function expecting one at a time, like $\lambda x.\ \mathbf{if}\ x == 0\ \mathbf{then}(\lambda y.y)\ \mathbf{else}(\lambda y.y/x)$. Call-By-Unboxed-Value foundation naturally has the tools to spell out and these different calling conventions. In fact, the separate run-time actions of (1) allocating a closure on a heap, (2) calling a closure, (3) delaying a function call until the function code is calculated, and (4) popping the next frame off the stack are all expressed by

separate syntactic forms, and reflected in the type system, giving fine-grain control over closures allocation and function calls that is safe across function and module boundaries.

In developing the Call-By-Unboxed-Value paradigm, we make the following contributions:

- Section 3 defines the Call-By-Unboxed-Value λ-calculus, its syntax, type-and-kind system, operational semantics, and equational theory.
- Section 4 presents examples using Call-By-Unboxed-Value to explicate run-time details of functional programs. In particular, ordinary type polymorphism alone can already take advantage of representation irrelevance without abstracting over representations.[1]
- Section 5 shows how to embed the well-studied Call-By-Push-Value [31] into Call-By-Unboxed-Value, and proves that a polymorphic Call-By-Push-Value corresponds (in types and equality) to a Call-By-Unboxed-Value encoding of uniform representation.
- Section 6 gives a low-level abstract machine where representations map to different types of registers, and the boxing and unboxing primitives map to read and write operations in a global store. With this, we show how to compile and run (untyped) Call-By-Unboxed-Value and prove correspondence between both their operational semantics and type systems.

2 KEY IDEAS: THE ADVANTAGE OF BEING SECOND-CLASS

Avoid lifting at all costs. The first semantic analysis of unboxed values [47] observed that they *must* be evaluated first before they can be passed to functions or bound to variables. Delayed arguments are compiled as *thunks*—addresses to code that can generate their value on-demand—represented by pointers. A thunk pointer cannot be stored in a floating-point register, so even a lazy language needs to make sure unboxed arguments are passed strictly by value.

Elegantly, the indirection cost of a lazy floating-point number is reflected in denotational semantics: the domain of efficient unboxed numbers must be *unlifted*. So for an efficient implementation, we need a semantics lets us avoid lifting as much as possible, such as Call-By-Push-Value [31] which avoids implicit lifts since they are easy to add but hard to remove. This is achieved by separating values that *already are* versus computations that *will do* as two different kinds of types:

$$\mathit{ValueType} \ni A ::= A_0 \times A_1 \mid A_0 + A_1 \mid \mathrm{U}\,\underline{B} \qquad \mathit{ComputationType} \ni \underline{B} ::= A \to \underline{B} \mid \underline{B}_0 \,\&\, \underline{B}_1 \mid \mathrm{F}\,A$$

Costly lifts only happen in the explicit transitions ($\mathrm{U}\,\underline{B}$ and $\mathrm{F}\,A$) between values and computations.

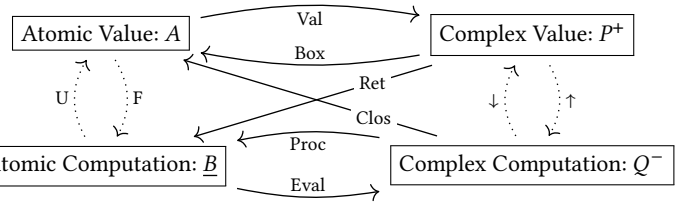
This arrangement is no accident, appearing again in the Calculus of Unity [55]—an interpretation of proof-theoretic *focusing* [3, 29] as pattern matching—for completely different reasons. A key step of focusing is to recognize *positive* versus *negative* types, divided like so:

$$\mathit{PositiveType} \ni P^+ ::= P_0^+ \otimes P_1^+ \mid P_0^+ \oplus P_1^+ \mid {\downarrow}Q^- \qquad \mathit{NegativeType} \ni Q^- ::= P^+ \to Q^- \mid Q_0^- \,\&\, Q_1^- \mid {\uparrow}P^+$$

Interesting. The two foundations arose for different reasons, but make identical divisions: value types seem positive, and computation types seem negative. So the two are the same, right?

Disagreements on who is first-class. The parallel seems to line up perfectly in many ways. Value and positive types model call-by-value whereas computation and negative types model call-by-name. Value and positive types model data types whereas computation and negative types model things like functions. Surprisingly, there is one glaring exception: they *disagree* on first-class status. Only values can be named in Call-By-Push-Value. With focusing, interesting positive values are second class: they cannot be given *one* name, because the program can—and *must*—deconstruct them.

[1] This is not to say there would be anything wrong with adding kind or representation polymorphism, but rather the design of the Call-By-Unboxed-Value λ-calculus seems to be able to handle the motivating examples already. If polymorphism over kinds is desired anyway, we expect no special difficulty in adding it.



$\mathrm{F}\,A = \mathrm{Ret}(\mathrm{Val}\,A)$ $\quad \mathrm{U}\,\underline{B} = \mathrm{Clos}(\mathrm{Eval}\,\underline{B})$ $\quad \uparrow P^+ = \mathrm{Eval}(\mathrm{Ret}\,P^+)$ $\quad \downarrow Q^- = \mathrm{Val}(\mathrm{Clos}\,Q^-)$

Fig. 1. The four kinds of types, and embeddings between them (dotted arrows are derived from solid ones).

Our main idea is to combine these two similar systems while respecting their disagreement about first-class status. Like Call-By-Push-Value, a variable always denotes an unknown value. Like focusing, pattern matching is mandatory, and data structures cannot be named. The key to satisfying both constraints at once was already hinted at in [55]. To account for machine primitives like numbers, the Calculus of Unity has special exceptions for "atomic" positive types with no known structure in the language, but since their structure is unknown, they are always just an unhelpfully generic "x." What if we could talk about what goes on inside atomic values, too?

The result is Call-By-Unboxed-Value. It splits programs twice between two orthogonal dimensions: value versus computation, and atomic versus complex. The atomic half of Call-By-Unboxed-Value corresponds to Call-By-Push-Value, wherein values are simple to name (representing machine primitives like numbers and pointers) and computations are ready to run (needing only a pointer to the top of a call stack, or nothing at all). The complex half of Call-By-Unboxed-Value corresponds to focusing and describes unboxed data structures and multi-part calling conventions. There is no limit to how many registers an unboxed data structure can occupy, which is *why* pattern matching is mandatory. Matching on a tuple (x, y, z) is the instruction for moving the three separate atoms into the three registers named x, y, and z. Dually, complex computations denote code that is *not* ready to run without more information, such as a function that needs arguments to safely call.

"Here" versus "there": Why two dimensions are better than one. The two-dimensional division of programs is illustrated in fig. 1, along with the transition between each quadrant. Solid arrows denote primitive operations within Call-By-Unboxed-Value; dotted arrows are derived and correspond to ones found in Call-By-Push-Value and the Calculus of Unity. While the twofold division creates more modes of transition, each one has a single, familiar, operational significance. By more finely decomposing the complex dotted arrows, the primitive transitions can be combined in new ways that are familiar in low-level programs but couldn't be explicated in either system.

The top row is concerned with values. Of course, atomic values like integers and pointers can be stored in a larger complex data structure, signaled by Val. But to go the other way, a complex data structure—which might bring together an multiple registers and a tag to describe its shape—cannot just be stuffed in one register. Instead, it has to be Boxed by storing its information in memory and then using an atomic pointer to it instead. Similarly, the bottom is concerned with computations. A complex computation may need many immediate inputs in registers to run correctly, but an atomic computation just wants something simple like a pointer to the call stack. Eval punctuates the end of a complex computation's input, giving single action to evaluate. Proc *boxes* a complex calling context—pushing a new frame on the stack—and runs an atomic action with the new stack. The diagonal arrows are the only transitions between values and computations. Ret describes an atomic computation that is ready to run, eventually returning multiple results in registers (a complex value). Likewise, Clos describes an atomic pointer value to a closure around a complex computation.

```
quotRem :: Nat → Nat → (Nat × Nat)                                   -- Haskell-like, functional style
quotRem x y | x < y      = (0, x)
            | otherwise = let (q, r) = quotRem (x − y) y in (1 + q, r)

quotRem : Nat → Nat → F(Nat × Nat)                                   -- Call-by-push-value
quotRem = λx.λy. do b ← x < y;
                 match b as { True  → return (0, x)
                            False → do x′ ← x − y;
                                       do z ← quotRem x′ y;
                                       match z as (q, r) → do q′ ← 1 + q;
                                                              return (q′, r) }

quotRem : Val Nat → Val Nat → Eval(Ret(Val Nat × Val Nat))           -- Call-by-unboxed-value
quotRem = { val int x · val int y · eval sub → do x < y as {
                 1, () → ret (val 0, val x)                          -- true case
                 0, () → do  val int x′ ← x − y;                     -- false case
                             do (val int q, val int r) ← quotRem (val x′) (val y). eval sub;
                             do  val int q′ ← 1 + q;
                             ret (val q′, val r) }}
```

Fig. 2. The same numeric algorithm in functional style, call-by-push-value, and call-by-unboxed-value.

In contrast, the two columns correspond to the two inspirational calculi: Call-By-Push-Value on the left and Calculus of Unity on the right. Notice that, while the U and F transitions and ↓ and ↑ polarity shifts can be faithfully derived from the other ones, but not vice versa. Round trips via Val and Box let us describe the details of pointer indirection to fully-evaluated data structures, like linked lists, without adding laziness. Call-By-Push-Value or focusing on their own do not distinguish between "here" and "there," but they can when they are put together in Call-By-Unboxed-Value.

A first taste of call-by-unboxed-value. To get an initial impression of what call-by-unboxed-value programs look like, we present an example function for simultaneously calculating the quotient and remainder of two numbers at the same time in fig. 2. First, the function is presented in a familiar, Haskell-like syntax. Next, we show the translation into call-by-push-value which brings out details of its step-by-step execution. Namely, the result of each operation — like an arithmetical operator or function call — is named in a sequence of steps annotated by the **do** keyword, reminiscent of monadic **do**-notation, and the final result is given by an explicit **return** statement represented by the F in the function's type. Additionally, pattern-matching or branching is represented as a separate **match** statement. Despite explicating these details, more still remain. Are the returned pairs (q', r) and $(0, x)$ allocated on the heap? Are the function arguments passed one at a time (requiring a closure to be allocated and consumed in each recursive loop)? These are left open-ended.

These kinds of questions are answered by the final call-by-unboxed-value version. The variable bindings of the form val int x denotes a named value stored in an integer-sized register (or on the stack, if all integers are full). The result bound by a **do** *must* be immediately matched on, including destructuring a tuple (as in the recursive result (val int q, val int r)) or choosing a response (as in the boolean branches for 1, () representing True and 0, () representing False). This means the pair **ret** (val q', val r) is returned unboxed, without allocation. Additionally, applying a function *never* triggers evaluation on its own; that second action is explicated by the "eval sub" operation that evaluates a subroutine. So from its syntax, we know the call-by-unboxed quotRem will never touch the heap, and will only push a single return pointer on the stack for each recursive call.[2]

[2]This, too, could be eliminated by rewriting the function in accumulator style so the recursive call to *quotRem* is the final tail call. Doing so would syntactically guarantee that the function is implemented as a loop that runs in constant space.

Syntax of complex values and complex computations:

$$\begin{array}{rlrl} \mathit{StructShape} \ni s & ::= () \mid s_0, s_1 \mid b, s \mid \Box, s \mid \mathrm{val}\,\Box & \mathit{StackShape} \ni k & ::= s \cdot k \mid b \cdot k \mid \Box \cdot k \mid \mathrm{eval}\,O \\ \mathit{Struct} \ni S & ::= s[V\ldots] & \mathit{Stack} \ni K & ::= k[V\ldots] \\ \mathit{Pattern} \ni p & ::= s[R\,x : A\ldots] & \mathit{Copattern} \ni q & ::= k[R\,x : A\ldots] \\ \mathit{MatchCode} \ni G & ::= \{\, p \to M\ldots \,\} \mid g & \mathit{FunCode} \ni F & ::= \{\, q \to M\ldots \,\} \mid f \\ \mathit{Bit} \ni b & ::= \mathtt{0} \mid \mathtt{1} & \mathit{Call} \ni L & ::= \lambda F \mid M.\,\mathrm{enter} \mid V.\,\mathrm{call} \end{array}$$

Syntax of atomic values and computations:

$$\begin{array}{rlrl} \mathit{Value} \ni V & ::= R\,x \mid \mathrm{box}\,S \mid \mathrm{clos}\,F \mid n \mid n.n \mid T & \mathit{Rep} \ni R & ::= \mathrm{ref} \mid \mathrm{int} \mid \mathrm{flt} \mid \mathrm{ty} \\ \mathit{Comp} \ni M & ::= S\,\mathbf{as}\,G \mid \mathbf{unbox}\,V\,\mathbf{as}\,G \mid \mathbf{do}\,M\,\mathbf{as}\,G & \mathit{Obs} \ni O & ::= \mathrm{run} \mid \mathrm{sub} \\ & \quad \mid \mathbf{ret}\,S \mid \mathrm{proc}\,F \mid \langle L \parallel K \rangle & & \end{array}$$

Syntax of types:

$$\begin{array}{rl} \mathit{Type} \ni T & ::= A \mid B \mid P \mid Q \\ \mathit{Kind} \ni \tau & ::= R\,\mathbf{val} \mid \mathbf{cplx\ val} \mid O\,\mathbf{comp} \mid \mathbf{cplx\ comp} \\ \mathit{CmplxValTy} \ni P & ::= x \mid 1 \mid P_0 \times P_1 \mid 0 \mid P_0 + P_1 \mid \exists R\,x : A.\,P \mid \mathrm{Val}\,A \\ \mathit{AtomValTy} \ni A & ::= x \mid \mathrm{Box}\,P \mid \mathrm{Clos}\,Q \mid \mathrm{Int} \mid \mathrm{Nat} \mid \mathrm{Float} \mid \mathrm{Type}\,\tau \\ \mathit{CmplxCompTy} \ni Q & ::= x \mid P \to Q \mid \top \mid Q_0 \,\&\, Q_1 \mid \forall R\,x : A.\,Q \mid \mathrm{Eval}\,B \\ \mathit{AtomCompTy} \ni B & ::= x \mid \mathrm{Ret}\,P \mid \mathrm{Proc}\,Q \mid \mathrm{Void} \end{array}$$

Fig. 3. The syntax of the Call-By-Unboxed-Value λ-calculus.

$$\begin{array}{rlrl} \langle L\ S \parallel K \rangle & = \langle L \parallel S \cdot K \rangle & \langle L\ b \parallel K \rangle & = \langle L \parallel b \cdot K \rangle \\ \langle L\ V \parallel K \rangle & = \langle L \parallel V \cdot K \rangle & L.\,\mathrm{eval}\,O & = \langle L \parallel \mathrm{eval}\,O \rangle \\ \multicolumn{2}{c}{\mathbf{do}\,p \leftarrow M; N = \mathbf{do}\,M\,\mathbf{as}\,\{\, p \to N \,\}} & \multicolumn{2}{c}{\mathbf{unbox}\,p \leftarrow V; N = \mathbf{unbox}\,V\,\mathbf{as}\,\{\, p \to N \,\}} \end{array}$$

Fig. 4. Syntactic sugar for writing call stacks in functional style and single-case matching.

3 CALL-BY-UNBOXED-VALUE λ-CALCULUS

We now present the polymorphic Call-By-Unboxed-Value λ-calculus: its syntax (section 3.1), operational semantics (section 3.2), type system (section 3.3), and equational theory (section 3.4). Peculiarly, functions are called with complex unboxed data structures as parameters, and yet these very unboxed structures are second-class citizens that cannot be directly named. Reconciling these two seemingly contrary design decisions makes this calling convention useful for combining both polymorphism with multiple kinds of atomic value representations.

3.1 Syntax

The Call-By-Unboxed-Value λ-calculus's syntax is given in fig. 3. Being based on the λ-calculus, it is not as perfectly symmetric as something like the Calculus of Unity [55]; nevertheless, we attempt to present it in a way that highlights the implicit dualities that are present, as well as to eliminate unnecessary redundancies whenever possible. To aid in writing examples, we use syntactic sugar to write structures, stacks, patterns, and copatterns inline, in the usual way. For example, instead of $(1, \mathrm{val}\,\Box, \mathrm{val}\,\Box)[\mathrm{int}\,x, 3.14]$, we will write $(1, \mathrm{val\,int}\,x, \mathrm{val}\,3.14)$. We also sometimes use syntactic sugar given in fig. 4 to write curried function applications in the more familiar λ-calculus style, or to list out a nested chain of single-case, pattern-matching bindings as a sequence of steps like fig. 2.

Complex structures (s, S, p, G). Every complex data structure has a particular *shape* that describes how it was constructed out of atomic parts. As such, a structure shape s is a context where constructors surround multiple holes $\square$ where atomic values can be inserted. Many of these constructors are familiar: an empty tuple $()$, a pair (s_0, s_1), an injection (b, s) into the sum type $P_0 + P_1$ where b is a $\mathtt{0}$ or $\mathtt{1}$ *bit*. We also have the base case $\mathrm{val}\ \square$ of type $\mathrm{Val}\ A$ where an atomic value (of type A) is inserted, as well as the modular (*i.e.,* existential $\exists$) package $\square, s$ of type $\exists R\ x : A.\ P$ in which an atomic value (importantly, a type) is named x and can be mentioned in s's type.

Actual concrete structures S are introduced by filling all $\square$'s with real values, written as $s[V\ldots]$, and are eliminated by pattern matching. Patterns p are formed by filling a shape with distinct variables $s[R\ x : A\ldots]$ annotated by their representations and types; we may omit these annotations when they are clear from context or unneeded. Pattern-matching code G is a set of alternatives $\{\ p \to M\ldots\ \}$ sending patterns to an atomic computation M, or else some primitive operation g.

Complex call stacks (k, K, q, F). Every complex computation must be executed in a context with a very specific shape, taking the form of a call stack. Like structures, complex call stack shapes k are multi-holed contexts where each hole $\square$ surrounds an atomic value. Possible call stack shapes include an unboxed function call $s \cdot k$, in which s is the argument's shape and k specifies the rest of the call, and a projection $b \cdot k$ out of a binary product $Q_0\ \&\ Q_1$ in which b says which option k calls. Polymorphic (*i.e.,* universal) specialization $\square \cdot k$ of type $\forall R\ x : A.\ Q$ names the value (*e.g.,* a type) placed in the $\square$ as x in the type of k. $\mathrm{eval}\ O$ marks a finished calling context that can now be evaluated. The annotation O describes the context of evaluation: will it run as a sub-computation of the larger program (sub) and return to some caller, or is it "naked" and running with no larger context (run). Like structures, call stacks K are built by filling the stack shapes with values, written $k[V\ldots]$, and used by *copattern matching* [1]. Copatterns q fill a stack shape with distinct variables $k[R\ x : A\ldots]$. *Copattern-matching code* F, *i.e.,* function code, is a set of alternatives $\{\ q \to M\ldots\ \}$ sending copatterns to atomic computations M, or some primitive function f.

Atomic values (V, R). Atomic values have simple enough run-time representations to store directly in a register, like a number. Each atomic value V has a self-evident representation R, spelling out the low-level details needed to implement operations on values. Both signed (Int) and unsigned (Nat) whole numbers n are represented as int, and a floating-point number $n.n$ is represented as flt. Some values are represented as references (ref) into long-term storage, including the boxed complex structures ($\mathrm{box}\ S$ of type $\mathrm{Box}\ P$) or closure around function code ($\mathrm{clos}\ F$ of type $\mathrm{Clos}\ Q$).

We also admit types T as atomic to be used as parameters for polymorphism $\forall\ \mathrm{ty}\ x{:}\mathrm{Type}\ \tau.\ Q$ à la System F [22, 23] and modular packages $\exists\ \mathrm{ty}\ x{:}\mathrm{Type}\ \tau.\ P$—this is a syntactic convenience used in practice by compilers like GHC to easily include types in the list of function parameters, instead of $k[T\ldots, V\ldots]$. But to be sure, these type parameters should still be erasable because they never impact run-time behavior (see section 6.4). Thus, we give type values the representation ty, with the understanding that they occupy "phantom" registers that don't really exist in a real machine.

The only atomic value left is a variable, which reads a value already stored in a register. Variables are the only form of value whose representation is not immediately obvious: x could be assigned a reference or a number. Therefore, we annotate variable access with its representation as $R\ x$ (omitted when clear from context), to distinguish different instructions like $\mathrm{ref}\ x$ for reading an address register named x or $\mathrm{flt}\ x$ for reading a floating-point register coincidentally named x, too.

Atomic computations (M, O). The last group of syntax involves atomic computations that can just run on their own accord without referencing their context. The computation $S\ \mathbf{as}\ G$ matches the structure S against the patterns of G. But notice how trivial this is: S must already be a fully-built structure made with known constructors that have to exactly match against the patterns in G. In

Operational reduction rules $\boxed{M \mapsto M'}$

$$(\beta\,\text{Box}) \quad \mathbf{unbox}\,\text{box}\,S\,\mathbf{as}\,G \mapsto S\,\mathbf{as}\,G \qquad (\beta\,\text{Clos}) \quad \langle \text{clos}\,F.\,\text{call} \,\|K\rangle \mapsto \langle \lambda F \parallel K\rangle$$

$$(\beta\,\text{Ret}) \quad \mathbf{do}\,\mathbf{ret}\,S\,\mathbf{as}\,G \mapsto S\,\mathbf{as}\,G \qquad (\beta\,\text{Proc}) \quad \langle \text{proc}\,F.\,\text{enter} \,\|K\rangle \mapsto \langle \lambda F \parallel K\rangle$$

$$(Prim_g) \quad S\,\mathbf{as}\,g \mapsto g(S) \qquad (Prim_f) \quad \langle \lambda f \parallel K\rangle \mapsto f(K)$$

$$(\beta\,\mathbf{as}) \quad s'[V_i^{\,i\in I}]\,\mathbf{as}\,\{\,s[R_{si}\,x_{si} : A_{si}^{\,i\in I}] \to M_s^{\,s\in P}\,\} \mapsto M_{s'}[V_i/(R_{s'i}\,x_{s'i})^{\,i\in I}] \qquad (s' \in P)$$

$$(\beta\lambda) \quad \langle \lambda\,\{\,k[R_{ki}\,x_{ki} : A_{ki}^{\,i\in I}] \to M_k^{\,k\in Q}\,\} \parallel k'[V_i^{\,i\in I}]\rangle \mapsto M_{k'}[V_i/(R_{k'i}\,x_{k'i})^{\,i\in I}] \qquad (k' \in Q)$$

Evaluation contexts (E) and terminal forms:

$$EvalCxt \ni E ::= \square \mid \mathbf{do}\,E\,\mathbf{as}\,G \mid \langle E.\,\text{enter} \,\|K\rangle$$

$$\frac{M \mapsto M}{E[M] \mapsto E[M']} \qquad \frac{g(S) = \text{terminal}}{S\,\mathbf{as}\,g\,\text{terminal}} \qquad \frac{f(K) = \text{terminal}}{E[\langle \lambda f \parallel K\rangle]\,\text{terminal}}$$

Fig. 5. The Call-By-Unboxed-Value operational semantics.

effect, every $S\,\mathbf{as}\,G$ expression can either be statically resolved, as-is, or it never will, independent of its context. For example, S might refer to some free variables, but their values will *never* be relevant for deciding the branch in $S\,\mathbf{as}\,G$. Instead, loading the contents of a boxed data structure is accomplished *exclusively* by $\mathbf{unbox}\,V\,\mathbf{as}\,G$, which immediately deconstructs its shape.

We still need to sequence sub-computations and remember the results they return. Call-By-Push-Value does this with a $\mathbf{do}\,x \leftarrow M; M'$ computation which runs M until it returns a result named x before continuing to M'. Call-by-Unboxed-Value has a similar atomic computation $\mathbf{do}\,M\,\mathbf{as}\,G$ with one key difference: the sub-computation returns M an *unboxed* result that is matched in place. The unboxed S returned by $\mathbf{ret}\,S$ is a second-class entity that cannot be named directly, since can contain *multiple values* take *many different shapes*. Therefore, a **do**-statement is forced to immediately pattern-match on the result to name the atomic values and decide how to continue.

Finally, we need a way to operate with function code. A fully applied function call can be written as $\langle L \parallel K\rangle$ where K is the complete call stack and L describes how the call is initiated, either: (1) directly invoking known code as λF, (2) calling a first-class closure object as $V.\,\text{call}$, or (3) running a second-class procedure as $\text{proc}\,F$. Procedures are useful when function is being used immediately (so no closure is allocated), but its code is not yet known and needs to be computed. To do so, K is put aside into long-term storage as a frame on the call stack until the computation finishes, yielding $\text{proc}\,F$ which pops that frame off the stack and continues as F.

3.2 Operational semantics

The Call-By-Unboxed-Value operational semantics is given in fig. 5, containing only eight reduction rules, many similar to one another. β Box and β Ret handle unboxing and returning, respectively; both dissolve into a known pattern match $S\,\mathbf{as}\,G$. Likewise, β Clos and β Proc handle calling a closure and entering a computed sub-procedure, respectively, via a known function call $\langle \lambda F \parallel K\rangle$.

All that remains is to reduce these statically-known forms of (co)pattern matching. With the shape of a complex structure and the matching code at hand, $\beta\,\mathbf{as}$ just looks up the chosen shape among the alternatives and substitutes the contained atomic values for the variables bound by the matching pattern, continuing as associated computation. $\beta\lambda$ works in much the same way by comparing stack shapes to choose a branch and substituting atomic values for local variables to run the associated response. Note that substitution is only defined for values and variables of the same representation. So for example, $M[3.14/\text{flt}\,x]$ is defined, as is $M[\text{flt}\,y/\text{flt}\,x]$, but $M[\text{clos}\,F/\text{flt}\,x]$ and $M[\text{ref}\,y/\text{flt}\,x]$ are *undefined* and give no result, since addresses don't fit into floating-point registers.

Primitive operations. The last two reduction rules cover primitive operations f and g meant to express instructions of the machine for built-in atomic types like int and flt. Each primitive operation needs to be given a specification for what it does on structures, written $g(S)$, or stacks, written $f(K)$. Here are some examples of primitive arithmetic:

$$\begin{aligned} \mathit{eqint\#}(\mathrm{val}\, n \cdot \mathrm{val}\, n \cdot \mathrm{eval}\, \mathrm{sub}) &= \mathbf{ret}\, 1, () \\ \mathit{eqint\#}(\mathrm{val}\, n \cdot \mathrm{val}\, n' \cdot \mathrm{eval}\, \mathrm{sub}) &= \mathbf{ret}\, \mathtt{0}, () \qquad (n \neq n') \\ \mathit{sqrt\#}(\mathrm{val}\, n.n \cdot \mathrm{eval}\, \mathrm{sub}) &= \mathbf{ret}\, \mathrm{val}\, \sqrt{n.n} \end{aligned}$$

where the equality check *eqint#* encodes the boolean result as an *unboxed* $1 + 1$. Some cases of a primitive operation might have no result, like division by zero, and must safely exit the program:

$$\begin{aligned} \mathit{divmod\#}(\mathrm{val}\, n \cdot \mathrm{val}\, n' \cdot \mathrm{eval}\, \mathrm{sub}) &= \mathbf{ret}\, \mathrm{val}\, d, \mathrm{val}\, r \qquad (d \times n' + r = n, \quad n' \neq 0) \\ \mathit{divmod\#}(\mathrm{val}\, n \cdot \mathrm{val}\, 0 \cdot \mathrm{eval}\, \mathrm{sub}) &= \mathrm{terminal} \end{aligned}$$

Specific applications that are expected might happen, and that safely exit the program instead of returning a result, are *terminal*. Notice, too, that this operation simultaneously returns two unboxed integers at once—the dividend and the remainder—if there is an answer.

A *terminal operation*, which is terminal on every defined result, can be used intentionally to model the final state where the program exits normally (*end#*) or abnormally (*error#*), like so:

$$\mathit{end\#}(\mathrm{val}\, n) = \mathrm{terminal} \qquad \mathit{error\#}(\mathrm{val}\, n \cdot B \cdot K) = \mathrm{terminal}$$

Note that *end#* takes a complex value, so it is primitive matching code that can be triggered in a program **do** M **as** *end#*. *end#* is just expecting to receive an integer (the exit code), and stops the program when there is nothing left to do. In contrast, *error#* is meant to be a polymorphic function (from the fact that it takes a type parameter ty a) that can be used *anywhere*, which is useful for (safely) aborting a program when some unexpected condition occurs.

Primitive parametricity. Primitive operations could be defined arbitrarily. To ensure they are reasonable in some sense, we assume they are *parametric* in both types and references: they can take types and references as parameters, but cannot read or write their contents or directly compare addresses. Formally, we express parametricity as equations letting us abstract out the specific contents of any type or reference passed to a primitive operation.

Assumption 3.1 (Primitive Parametricity). All primitive operations must satisfy these equalities:

$$\begin{aligned} g(S[T/\mathrm{ty}\, x]) &= g(S)[T/\mathrm{ty}\, x] & f(K[T/\mathrm{ty}\, x]) &= f(K)[T/\mathrm{ty}\, x] \\ g(S[\mathrm{box}\, S'/\mathrm{ref}\, x]) &= g(S)[\mathrm{box}\, S'/\mathrm{ref}\, x] & f(K[\mathrm{box}\, S'/\mathrm{ref}\, x]) &= f(K)[\mathrm{box}\, S'/\mathrm{ref}\, x] \\ g(S[\mathrm{clos}\, F/\mathrm{ref}\, x]) &= g(S)[\mathrm{clos}\, F/\mathrm{ref}\, x] & f(K[\mathrm{clos}\, F/\mathrm{ref}\, x]) &= f(K)[\mathrm{clos}\, F/\mathrm{ref}\, x] \end{aligned}$$

As a consequence, note that these equations imply that pointer equality is forbidden as a primitive operation. Suppose we had such an operation defined as:

$$\begin{aligned} \mathit{eq\#}(\mathrm{val}\, \mathrm{ref}\, x \cdot \mathrm{val}\, \mathrm{ref}\, x \cdot \mathrm{eval}\, \mathrm{sub}) &= \mathbf{ret}\, 1, () \\ \mathit{eq\#}(\mathrm{val}\, \mathrm{ref}\, x \cdot \mathrm{val}\, \mathrm{ref}\, y \cdot \mathrm{eval}\, \mathrm{sub}) &= \mathbf{ret}\, \mathtt{0}, () \qquad (x \neq y) \end{aligned}$$

Then the parametricity of references forces the following equations for an arbitrary reference value V (where $k = \mathrm{val}\, \mathrm{ref}\, \square \cdot \mathrm{val}\, \mathrm{ref}\, \square \cdot \mathrm{eval}\, \mathrm{sub}$):

$$\begin{aligned} \mathit{eq\#}(k[V,V]) &= \mathit{eq\#}(k[\mathrm{ref}\, x, \mathrm{ref}\, x])[V/\mathrm{ref}\, x] &&= (\mathbf{ret}\, 1, ())[V/\mathrm{ref}\, x] &&= \mathbf{ret}\, 1, () \\ \mathit{eq\#}(k[V,V]) &= \mathit{eq\#}(k[\mathrm{ref}\, x, \mathrm{ref}\, y])[V/\mathrm{ref}\, x][V/\mathrm{ref}\, y] &&= (\mathbf{ret}\, \mathtt{0}, ())[V/\mathrm{ref}\, x][V/\mathrm{ref}\, y] &&= \mathbf{ret}\, \mathtt{0}, () \end{aligned}$$

So pointer equality operations like *eq#* have to be barred since they would force invalid equivalences like $\mathbf{ret}\, 1, () = \mathbf{ret}\, \mathtt{0}, ()$. Likewise, type equality is forbidden as a primitive operation.

$$\mathit{ValueEnv} \ni \Gamma, \Delta ::= \bullet \mid \Gamma, R\ x : A \qquad \mathit{CompEnv} \ni \Phi ::= B : O\ \mathbf{comp}$$

Base types $\boxed{T : \tau}$

$$\mathrm{Int} : \mathrm{int}\ \mathbf{val} \qquad \mathrm{Nat} : \mathrm{int}\ \mathbf{val} \qquad \mathrm{Float} : \mathrm{flt}\ \mathbf{val} \qquad \mathrm{Type}\ \tau : \mathrm{ty}\ \mathbf{val}$$
$$1 : \mathbf{cplx}\ \mathbf{val} \qquad 0 : \mathbf{cplx}\ \mathbf{val} \qquad \top : \mathbf{cplx}\ \mathbf{comp} \qquad \mathrm{Void} : \mathrm{run}\ \mathbf{comp}$$

Kinds of types $\boxed{\Gamma \vdash T : \tau}$

$$\frac{}{\Gamma, \mathrm{ty}\ x : \mathrm{Type}\ \tau, \Gamma' \vdash x : \tau}\ \mathit{TyVar} \qquad \frac{T : \tau}{\Gamma \vdash T : \tau}\ \mathit{BaseTy}$$

$$\frac{\Gamma \vdash P : \mathbf{cplx}\ \mathbf{val}}{\Gamma \vdash \mathrm{Box}\ P : \mathrm{ref}\ \mathbf{val}}\ \mathrm{Box}\,T \qquad \frac{\Gamma \vdash Q : \mathbf{cplx}\ \mathbf{comp}}{\Gamma \vdash \mathrm{Clos}\ Q : \mathrm{ref}\ \mathbf{val}}\ \mathrm{Clos}\,T$$

$$\frac{\Gamma \vdash P : \mathbf{cplx}\ \mathbf{val}}{\Gamma \vdash \mathrm{Ret}\ P : \mathrm{sub}\ \mathbf{comp}}\ \mathrm{Ret}\,T \qquad \frac{\Gamma \vdash Q : \mathbf{cplx}\ \mathbf{comp}}{\Gamma \vdash \mathrm{Proc}\ Q : \mathrm{sub}\ \mathbf{comp}}\ \mathrm{Proc}\,T$$

$$\frac{\Gamma \vdash P_0 : \mathbf{cplx}\ \mathbf{val} \quad \Gamma \vdash P_1 : \mathbf{cplx}\ \mathbf{val}}{\Gamma \vdash P_0 \times P_1 : \mathbf{cplx}\ \mathbf{val}}\ \times T \qquad \frac{\Gamma \vdash P_0 : \mathbf{cplx}\ \mathbf{val} \quad \Gamma \vdash P_1 : \mathbf{cplx}\ \mathbf{val}}{\Gamma \vdash P_0 + P_1 : \mathbf{cplx}\ \mathbf{val}}\ +T$$

$$\frac{\Gamma \vdash A : R\ \mathbf{val} \quad \Gamma, R\ x : A \vdash P : \mathbf{cplx}\ \mathbf{val}}{\Gamma \vdash \exists R\ x : A.\ P : \mathbf{cplx}\ \mathbf{val}}\ \exists T \qquad \frac{\Gamma \vdash A : R\ \mathbf{val}}{\Gamma \vdash \mathrm{Val}\ A : \mathbf{cplx}\ \mathbf{val}}\ \mathrm{Val}\,T$$

$$\frac{\Gamma \vdash P : \mathbf{cplx}\ \mathbf{val} \quad \Gamma \vdash Q : \mathbf{cplx}\ \mathbf{comp}}{\Gamma \vdash P \to Q : \mathbf{cplx}\ \mathbf{comp}}\ {\to}T \qquad \frac{\Gamma \vdash Q_0 : \mathbf{cplx}\ \mathbf{comp} \quad \Gamma \vdash Q_1 : \mathbf{cplx}\ \mathbf{comp}}{\Gamma \vdash Q_0\ \&\ Q_1 : \mathbf{cplx}\ \mathbf{comp}}\ \&T$$

$$\frac{\Gamma \vdash A : R\ \mathbf{val} \quad \Gamma, R\ x : A \vdash Q : \mathbf{cplx}\ \mathbf{comp}}{\Gamma \vdash \forall R\ x : A.\ Q : \mathbf{cplx}\ \mathbf{comp}}\ \forall T \qquad \frac{\Gamma \vdash B : O\ \mathbf{comp}}{\Gamma \vdash \mathrm{eval}\ B : \mathbf{cplx}\ \mathbf{comp}}\ \mathrm{Eval}\,T$$

Fig. 6. The kinds of types and typing environments.

3.3 Type system

The Call-By-Unboxed-Value type system is given in figs. 6 to 8. The kinds of types are classified in fig. 6—all P : **cplx val** and Q : **cplx comp** types are just complex with no further specification, but atomic value types A are further separated by their representation R, written $A : R$ **val**, and atomic computation types B are separated by the observational context O, written $B : O$ **comp**. For example, both Int and Nat share the kind int **val**, since their values are represented as (respectively, signed or unsigned) integers, whereas Box P and Clos Q share the kind ref **val** since their values are references. For atomic computations, both Ret P and Proc Q are kinds of sub-computations, written sub **comp**, since they both need to interact with the top of the stack (either to return some value(s) to an evaluation context or to pop the stack frame off and run a procedure). void is the sole run **comp** type, which classifies computations that need no context because they never return.

The types of values are classified in fig. 7. One set of rules involve introducing various shapes of structures, written $\Gamma \mid \Delta \vdash s : P$; where Δ lists the types of atomic values that fit in s's holes, P is the type of structure built when those holes are filled, and Γ keeps track of any free type variables in Δ or P. Since the holes of s are only distinguished by position, the order of Δ matters. Individual atomic values can only be well-typed, written $\Gamma \vdash V : A : R$ **val**, when their type has a known representation. Note that the premise of the *Match* rule must check for *all* the possible patterns (*i.e.,* all the possible shapes, up to renaming the holes) of type P to ensure that every case is covered.

Structure shapes $\boxed{\Gamma \mid \Delta \vdash s : P\ ;}$

$$\frac{}{\Gamma \mid \bullet \vdash () : 1\ ;}\ 1I \qquad \frac{\Gamma \mid \Delta_0 \vdash s_0 : P_0\ ; \quad \Gamma \mid \Delta_1 \vdash s_1 : P_1\ ;}{\Gamma \mid \Delta_0, \Delta_1 \vdash s_0, s_1 : P_0 \times P_1\ ;}\ \times I$$

$$\frac{\Gamma \mid \Delta \vdash s_0 : P_0\ ;}{\Gamma \mid \Delta \vdash \mathtt{0}, s_0 : P_0 + P_1\ ;}\ +I_0 \qquad \frac{\Gamma \mid \Delta \vdash s_1 : P_1\ ;}{\Gamma \mid \Delta \vdash 1, s_1 : P_0 + P_1\ ;}\ +I_1 \qquad \text{No } 0I \text{ rules}$$

$$\frac{\Gamma, R\,x : A \mid \Delta \vdash s : P\ ;}{\Gamma \mid (R\,x : A, \Delta) \vdash (\square, s) : (\exists R\,x : A.\ P)\ ;}\ \exists I \qquad \frac{\Gamma \vdash A : R\,\mathbf{val}}{\Gamma \mid R\,x : A \vdash \mathrm{val}\,\square : \mathrm{Val}\,A\ ;}\ \mathrm{Val}\,I$$

Structures $\boxed{\Gamma \vdash S : P}$, patterns $\boxed{\Gamma \mid \Delta \vdash p : P\ ;}$, and pattern match $\boxed{\Gamma\ ;\ G : P \vdash \Phi}$

$$\frac{\Gamma \mid \Delta \vdash s : P\ ; \quad \Gamma \vdash V_i^{i \in I} \ldots : \Delta}{\Gamma \vdash s[V_i^{i \in I}\ldots] : P}\ \mathit{Struct} \qquad \frac{\Gamma \mid \Delta \vdash s : P\ ;}{\Gamma \mid \Delta \vdash s[\Delta] : P\ ;}\ \mathit{Pat}$$

$$\frac{\forall(\Gamma \mid \Delta_p \vdash p : P\ ;).\quad \Gamma, \Delta_p \vdash M_p : \Phi}{\Gamma\ ;\ \{\, p \to M_p^{p \in P}\ldots \,\} : P \vdash \Phi}\ \mathit{Match} \qquad \frac{g : P}{\Gamma\ ;\ g : P \vdash \mathrm{void} : \mathrm{run}\,\mathbf{comp}}\ \mathit{PrimMatch}$$

Atomic values $\boxed{\Gamma \vdash V : A : R\,\mathbf{val}}$

$$\frac{}{\Gamma, R\,x : A, \Gamma' \vdash R\,x : A : R\,\mathbf{val}}\ \mathit{Var} \qquad \frac{}{\Gamma \vdash n.n : \mathrm{Float} : \mathrm{flt}\,\mathbf{val}}\ \mathrm{Float}\,I$$

$$\frac{}{\Gamma \vdash n : \mathrm{Int} : \mathrm{int}\,\mathbf{val}}\ \mathrm{Int}\,I \qquad \frac{n \geq 0}{\Gamma \vdash n : \mathrm{Nat} : \mathrm{int}\,\mathbf{val}}\ \mathrm{Nat} \qquad \frac{\Gamma \vdash T : \tau}{\Gamma \vdash T : \mathrm{Type}\ \tau : \mathrm{ty}\,\mathbf{val}}\ \mathrm{Type}I$$

$$\frac{\Gamma \vdash S : P}{\Gamma \vdash \mathrm{box}\,S : \mathrm{Box}\,P : \mathrm{ref}\,\mathbf{val}}\ \mathrm{Box}I \qquad \frac{\Gamma \vdash F : Q\ ;}{\Gamma \vdash \mathrm{clos}\,F : \mathrm{Clos}\,Q : \mathrm{ref}\,\mathbf{val}}\ \mathrm{Clos}I$$

Value sequences $\boxed{\Gamma \vdash V\ldots : \Delta}$

$$\frac{}{\Gamma \vdash \bullet : \bullet} \qquad \frac{\Gamma \vdash V : A : R\,\mathbf{val} \quad \Gamma \vdash V'\ldots : \Delta[V/R\,x]}{\Gamma \vdash V, V'\ldots : (R\,x : A), \Delta}$$

Fig. 7. Types of complex and atomic values.

The types of computations are classified in fig. 8. We have rules for introducing various shapes of stacks, written $\Gamma \mid \Delta\ ;\ k : Q \vdash \Phi$ where Δ lists the types of atomic values that fit in k's holes, Q is the type of complex computation the stack can call to produce an atomic computation Φ, and Γ keeps track of free type variables in Δ, Q, or Φ. We follow Gentzen's tradition [20] and write $k : Q$ to the left of the $\vdash$, similar to [11, 53], since these rules correspond to the sequent calculus' left rules. As before, *CoMatch* requires *all* possible copatterns of type Q be covered. Atomic computations, $\Gamma \vdash M : \Phi$, can only be well-typed when we statically know how to obvserve them. For certain atomic computations, like $\mathbf{ret}\,S : \mathrm{Ret}\,P : \mathrm{sub}\,\mathbf{comp}$ and $\mathrm{proc}\,F : \mathrm{Proc}\,Q : \mathrm{sub}\,\mathbf{comp}$, this is fixed to sub, but the block forms like **do** and **unbox** could have any type of result with any observation.

Aside 3.2. Every complex value type P classifies a finite number (zero or more) of possible structure shapes s; likewise Q classifies a finite number of stack shapes k. As such, the number of premises to the *Match* and *CoMatch* rules can vary but is always finite. Moreover, polymorphism in P or Q can force their set of shapes to be zero if a generic type variable is seen before reaching something atomic. For example, $\mathrm{Val\,Float} \times \mathrm{Val\,Int}$ describes only the shape $(\mathrm{val}\,\square, \mathrm{val}\,\square)$ but $\exists\,\mathrm{ty}\,a : \mathrm{Type}\ \mathbf{cplx\ val}$. $\mathrm{Val\,Float} \times a$ describes *no shapes*, since a generic $\mathrm{ty}\,a : \mathbf{cplx\ val}$ has no known patterns. Some polymorphism—both atomic and complex—allows for pattern-matching, however. For

Stack shapes $\boxed{\Gamma \mid \Delta \; ; k : Q \vdash \Phi}$

$$\frac{\Gamma \mid \Delta \vdash s : P \quad \Gamma \mid \Delta' \; ; k : Q \vdash \Phi}{\Gamma \mid \Delta, \Delta' \; ; s \cdot k : P \to Q \vdash \Phi} \to L$$

$$\frac{\Gamma \mid \Delta \; ; k_0 : Q_0 \vdash \Phi}{\Gamma \mid \Delta \; ; \mathtt{0} \cdot k_0 : Q_0 \,\&\, Q_1 \vdash \Phi} \,\&L_0 \qquad \frac{\Gamma \mid \Delta \; ; k_1 : Q_1 \vdash \Phi}{\Gamma \mid \Delta \; ; \mathtt{1} \cdot k_1 : Q_0 \,\&\, Q_1 \vdash \Phi} \,\&L_1 \qquad \text{No } \top L \text{ rules}$$

$$\frac{\Gamma, R\, x : A \mid \Delta \; ; k : Q \vdash \Phi}{\Gamma \mid (R\, x : A, \Delta) \; ; (\square \cdot k) : (\forall R\, x : A.\; Q) \vdash \Phi} \,\forall L \qquad \frac{\Gamma \vdash B : O\, \mathbf{comp}}{\Gamma \mid \bullet \; ; \mathrm{eval}\, O : \mathrm{Eval}\, B \vdash B : O\, \mathbf{comp}} \,\mathrm{Eval}\, L$$

Stacks $\boxed{\Gamma \mid K : Q \vdash \Phi}$, copatterns $\boxed{\Gamma \mid \Delta \; ; q : Q \vdash \Phi}$, and function definitions $\boxed{\Gamma \vdash F : Q \; ;}$

$$\frac{\Gamma \mid \Delta \; ; k : Q \vdash \Phi \quad \Gamma \vdash V \ldots : \Delta}{\Gamma \mid k[V \ldots] : Q \vdash \Phi} \,\mathit{Stack} \qquad \frac{\Gamma \mid \Delta \; ; k : Q \vdash \Phi}{\Gamma \mid \Delta \; ; k[\Delta] : Q \vdash \Phi} \,\mathit{CoPat}$$

$$\frac{\forall (\Gamma \mid \Delta_q \; ; q : Q \vdash \Phi_q). \quad \Gamma, \Delta_q \vdash M_q : \Phi_q}{\Gamma \vdash \{\, q \to M_q{}^{q \in Q}_{\ldots} \,\} : Q \; ;} \,\mathit{CoMatch} \qquad \frac{f : Q}{\Gamma \vdash f : Q \; ;} \,\mathit{PrimFun}$$

Complex computation $\boxed{\Gamma \vdash L : Q}$

$$\frac{\Gamma \vdash F : Q \; ;}{\Gamma \vdash \lambda F : Q} \qquad \frac{\Gamma \vdash V : \mathrm{Clos}\, Q : \mathrm{ref}\, \mathbf{val}}{\Gamma \vdash V.\, \mathrm{call} : Q} \,\mathrm{Clos}\, E \qquad \frac{\Gamma \vdash M : \mathrm{Proc}\, Q : \mathrm{sub}\, \mathbf{comp}}{\Gamma \vdash M.\, \mathrm{enter} : Q} \,\mathrm{Proc}\, E$$

Atomic computations $\boxed{\Gamma \vdash M : \Phi}$

$$\frac{\Gamma \vdash S : P}{\Gamma \vdash \mathbf{ret}\, S : \mathrm{Ret}\, P : \mathrm{sub}\, \mathbf{comp}} \,\mathrm{Ret}\, I \qquad \frac{\Gamma \vdash M : \mathrm{Ret}\, P : \mathrm{sub}\, \mathbf{comp} \quad \Gamma \; ; G : P \vdash \Phi}{\Gamma \vdash \mathbf{do}\, M\, \mathbf{as}\, G : \Phi} \,\mathrm{Ret}\, E$$

$$\frac{\Gamma \vdash S : P \quad \Gamma \; ; G : P \vdash \Phi}{\Gamma \vdash S\, \mathbf{as}\, G : \Phi} \,\mathit{StructCut} \qquad \frac{\Gamma \vdash V : \mathrm{Box}\, P : \mathrm{ref}\, \mathbf{val} \quad \Gamma \; ; G : P \vdash \Phi}{\Gamma \vdash \mathbf{unbox}\, V\, \mathbf{as}\, G : \Phi} \,\mathrm{Box}\, E$$

$$\frac{\Gamma \vdash L : Q \quad \Gamma \mid K : Q \vdash \Phi}{\Gamma \vdash \langle L \parallel K \rangle : \Phi} \,\mathit{StackCut} \qquad \frac{\Gamma \vdash F : Q \; ;}{\Gamma \vdash \mathrm{proc}\, F : \mathrm{Proc}\, Q : \mathrm{sub}\, \mathbf{comp}} \,\mathrm{Proc}\, I$$

Fig. 8. Types of complex and atomic computations.

example, $\exists\, \mathrm{ty}\, a : \mathrm{Type}\, \mathrm{ref}\, \mathbf{val}\,.\ \mathrm{Val}\, \mathrm{Float} \times \mathrm{Val}\, a$ and $\exists\, \mathrm{ty}\, a : \mathrm{Type}\, \mathbf{cplx}\, \mathbf{val}\,.\ \mathrm{Val}\, \mathrm{Float} \times \mathrm{Val}(\mathrm{Box}\, a)$ both describe the same shape $(\square, \mathrm{val}\, \square, \mathrm{val}\, \square)$. The same scenario occurs in stack shapes, where $\mathrm{Val}\, \mathrm{Float} \to \mathrm{Eval}\, \mathrm{Void}$ describes $(\mathrm{val}\, \square \cdot \mathrm{eval}\, \mathrm{run})$, both $\forall\, \mathrm{ty}\, a : \mathrm{Type}\, \mathrm{sub}\, \mathbf{comp}\,.\ \mathrm{Val}\, \mathrm{Float} \to \mathrm{Eval}\, a$ and $\forall\, \mathrm{ty}\, a : \mathrm{Type}\, \mathbf{cplx}\, \mathbf{comp}\,.\ \mathrm{Val}\, \mathrm{Float} \to \mathrm{Eval}(\mathrm{Proc}\, a)$ describe $(\square \cdot \mathrm{val}\, \square \cdot \mathrm{eval}\, \mathrm{sub})$, but $\forall\, \mathrm{ty}\, a : \mathrm{Type}\, \mathbf{cplx}\, \mathbf{comp}\,.\ \mathrm{Val}\, \mathrm{Float} \to a$ describes *no shapes*. The second-class status of complex structures and call stacks automatically enforces the ad-hoc monomorphism restrictions imposed by [15, 19].

Aside 3.3. Note that the typing rules for $\exists I$, $\forall L$, and for value sequences $V, V' : (R\, x : A), \Delta$ make it appear that types could depend on *any* kind of atomic value. However, in reality, only meaningful dependencies are on ty-represented values — standing in for a type — which can be accessed via the *TyVar* rule. There are no other rules in fig. 6 that allow that free variables of other representations (int, flt, *etc.*) to actually appear in well-formed types. Vice versa, the only allowed use of a ty-value is as a parameter to $\forall\, \mathrm{ty}\, x : \mathrm{Type}\, \tau.Q$ or $\exists\, \mathrm{ty}\, x : \mathrm{Type}\, \tau.P$; there are no other operations on $\mathrm{Type}\, \tau$ values. As such, we could restrict $\forall$ and $\exists$ to exactly these special cases, and limit the appearance of types only to parameters of stacks or structures, and get a syntax that more closely resembles System F [22] — a familiar basis for typed intermediate languages — without any

$$
\begin{array}{rcl}
(\eta\,\mathrm{Proc}) & \mathrm{proc}\,\{\, q \to \langle M.\,\mathrm{enter}\,\|q\rangle \,\}^{q\in Q}_{\cdots} = M & (M : \mathrm{Proc}\,Q) \\
(\eta\,\mathrm{Clos}) & \mathrm{clos}\,\{\, q \to \langle V.\,\mathrm{call}\,\|q\rangle \,\}^{q\in Q}_{\cdots} = V & (V : \mathrm{Clos}\,Q) \\
(\eta\,\mathrm{Box}) & \mathbf{unbox}\,V\,\mathbf{as}\,\{\, p \to M[\mathrm{box}\,p/x] \,\}^{p\in P}_{\cdots} = M[V/x] & (V : \mathrm{Box}\,P) \\
(\eta\,\mathrm{Ret}) & \mathbf{do}\,M\,\mathbf{as}\,\{\, p \to E[\mathbf{ret}\,p] \,\}^{p\in P}_{\cdots} = E[M] & (M : \mathrm{Ret}\,P)
\end{array}
$$

Fig. 9. Extensional η axioms of the typed equational theory.

change of expressiveness. We avoid doing so because the extra restrictions further complicate the grammar of syntax without providing any extra benefits that cannot already be inferred as-is.

Type Safety. The only thing remaining is types for primitive operations. As these are defined outside of the calculus itself, we use an abstract notion to classify when they can be safely assigned a type.

Assumption 3.4 (Primitive Safety). $g : P$ implies that $\Gamma \vdash g(S) : \mathrm{void} : \mathrm{run}\,\mathbf{comp}$ or $g(S)$ terminal for every $\Gamma \vdash S : P$ such that Γ binds only ref or ty variables. Likewise, $f : Q$ implies that $\Gamma \vdash f(K) : \Phi$ or $f(K)$ terminal for every $\Gamma \mid K : Q \vdash \Phi$ such that Γ binds only ref or ty variables.

For example, some primitive operations defined in section 3.2 can be safely assigned these types:

$$
\begin{array}{rl}
\mathit{eqint\#} & : \mathrm{Val\,Int} \to \mathrm{Val\,Int} \to \mathrm{Eval}(\mathrm{Ret}(1+1)) \\
\mathit{divmod\#} & : \mathrm{Val\,Int} \to \mathrm{Val\,Int} \to \mathrm{Eval}(\mathrm{Ret}(\mathrm{Val\,Int} \times \mathrm{Val\,Int})) \\
\mathit{error\#} & : \mathrm{Val\,Int} \to \forall\,\mathrm{ty}\,a : \mathrm{Type}\,\mathbf{cplx\,comp}\,.\,a
\end{array}
$$

In *error#*'s type, after receiving an Int error code, it proceeds as *any type of complex computation*. That means *error#* can be asked to return any complex result by instantiating $a = \mathrm{Eval}(\mathrm{Ret}\,b)$ for an arbitrary $b : \mathrm{Type}\,\mathbf{cplx\,val}$. We can also instantiate a to a function $(P \to Q)$ or product $(Q \,\&\, Q')$ in case we need to signal an error during a complex computation.

Assuming all primitive operations are safe, the Call-By-Unboxed-Value λ-calculus is type safe.

Lemma 3.5 (Progress). *If* $\bullet \vdash M : \mathrm{void} : \mathrm{run}\,\mathbf{comp}$*, then either* $M \mapsto M'$ *or* M terminal.

Lemma 3.6 (Preservation). *If* $\Gamma \vdash M : B : O\,\mathbf{comp}$ *and* $M \mapsto M'$ *then* $\Gamma \vdash M' : B : O\,\mathbf{comp}$.

3.4 Equational Theory

If we want to reason extensionally about program equality—based only on their input-output behavior—then we need some additional rules stating that taking things apart and putting them back together is unobservable. We only need four rules, written as familiar η-style axioms of the λ-calculus, are given in fig. 9. With them, the Call-By-Unboxed-Value equational theory is defined as the reflexive, transitive, symmetric, and compatible closure of these β and η rules (figs. 5 and 9).

Although we only list four extensional η-axioms, other familiar properties are derivable from them. The **do** identity η axiom and commuting conversions are both derivable:

$$
\begin{array}{rcl}
(\eta\,\mathrm{Ret}_{Id}) & \mathbf{do}\,M\,\mathbf{as}\,\{\, p \to \mathbf{ret}\,p \,\}^{p\in P}_{\cdots} = M \\
(cc\,\mathrm{Ret}) & E[\mathbf{do}\,M\,\mathbf{as}\,\{\, p \to M' \,\}^{p\in P}_{\cdots}] = \mathbf{do}\,M\,\mathbf{as}\,\{\, p \to E[M'] \,\}^{p\in P}_{\cdots}
\end{array}
$$

$\eta\,\mathrm{Ret}_{Id}$ is just a special case of $\eta\,\mathrm{Ret}$, and $E[\mathbf{do}\,\square\,\{\, p \to M' \,\}^{p\in P}_{\cdots}]$ is another evaluation context, so

$$
\begin{aligned}
E[\mathbf{do}\,M\,\mathbf{as}\,\{\, p \to M' \,\}^{p\in P}_{\cdots}] &=_{\eta\,\mathrm{Ret}} \mathbf{do}\,M\,\mathbf{as}\,\{\, p \to E[\mathbf{do}\,\mathbf{ret}\,p\,\mathbf{as}\,\{\, p \to M' \,\}^{p\in P}_{\cdots}] \,\}^{p\in P}_{\cdots} \\
&=_{\beta\,\mathrm{Ret}} \mathbf{do}\,M\,\mathbf{as}\,\{\, p \to E[M'] \,\}^{p\in P}_{\cdots}
\end{aligned}
$$

4 EXAMPLES OF REPRESENTATION IRRELEVANCE IN CALL-BY-UNBOXED-VALUE

We now turn to some examples of writing some simple polymorphic code into Call-By-Unboxed-Value, both to get familiar with its differences to high-level functional code, as well as to explore ways in which it can *already* express the representation-polymorphic code of [15, 19] without the need to fully characterize complex representations or to abstract over them.

The humble identity function. Let's start with the simplest possible example: the polymorphic identity function: $id\ x = x$. It is no surprise that this function can't *really* be polymorphic over different representations of x; eventually its machine code will hard-wire details about moving x around. For example, here are two hard-wired choices for fixing a's representation:

$$\begin{aligned}
&idFlt &&: \mathrm{Val\,Float} \to \mathrm{Eval}(\mathrm{Ret}(\mathrm{Val\,Float}))\\
&idFlt &&= \{\ \mathrm{val\,flt}\ x : \mathrm{Float} \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathrm{val\,flt}\ x\ \}\\
&idIntFlt &&: \forall\ \mathrm{ty}\ a : \mathrm{Type\,int}\ \mathbf{val}\,.\ \mathrm{Val}\ a \times \mathrm{Val\,Float} \to \mathrm{Eval}(\mathrm{Ret}(\mathrm{Val}\ a \times \mathrm{Val\,Float}))\\
&idIntFlt &&= \{\ \mathrm{ty}\ a \cdot (\mathrm{val\,int}\ x : a, \mathrm{val\,flt}\ y : \mathrm{Float}) \cdot \mathrm{eval\,sub} \to \mathbf{ret}(\mathrm{val\,int}\ x, \mathrm{val\,flt}\ y)\ \}
\end{aligned}$$

so that calling $\langle \lambda\, idFlt \parallel \mathrm{val}\ 3.14 \cdot \mathrm{eval\,sub}\rangle$ successfully matches the copattern, which computes to **ret** val 3.14, but $\langle \lambda\, idFlt \parallel (\mathrm{val}\ 5, \mathrm{val}\ 3.14) \cdot \mathrm{eval\,sub}\rangle$ is intuitively *not OK*, and this intuition is supported by the fact that the copattern does not match causing the computation to get stuck here. Likewise, the second specialization can be passed an unboxed pair with $\mathrm{Nat} \cdot (\mathrm{val}\ 2, \mathrm{val}\ 1.41) \cdot \mathrm{eval\,sub}$, since Nat is represented by int, but a call stack with a single floating-point value doesn't match.

However, trying to write a fully general function of type $\forall\ \mathrm{ty}\ a : \mathrm{Type}\ \mathbf{cplx\ val}\ .a \to \mathrm{Eval}(\mathrm{Ret}\ a)$ would fail—not from some arbitrary restriction, but simply because we don't know any patterns of a generic $\mathrm{ty}\ a : \mathbf{cplx\ val}$; it's not an atomic value so we cannot name it as $\mathrm{val}\ x$, and the type-specific rules don't apply. Instead, the most general-purpose identity function takes an atomic reference:

$$\begin{aligned}
&idRef &&: \forall\ \mathrm{ty}\ a : \mathrm{Type\,ref}\ \mathbf{val}\,.\ \mathrm{Val}\ a \to \mathrm{Eval}(\mathrm{Ret}(\mathrm{Val}\ a))\\
&idRef &&= \{\ \mathrm{ty}\ a \cdot \mathrm{val\,ref}\ x : a \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathrm{val\,ref}\ x\ \}
\end{aligned}$$

idRef can take all kinds of values at the usual cost of indirection. For example, *idRef* can be given the argument $\mathrm{box}(\mathrm{val}\ -4)$, $\mathrm{box}(\mathrm{val}\ 3.14)$, or $\mathrm{box}(\mathrm{val}\ 2, \mathrm{val}\ 1.41)$ (by instantiating a to Box Val Int, Box Val Float, and $\mathrm{Box}(\mathrm{Val\,Nat} \times \mathrm{Val\,Float})$, respectively). We can also pass closures around code to *idRef*, like clos *idRef* itself, since a $\mathrm{Clos}(\dots)$ is also an atomic reference value.

The fact that we can pass closures to *idRef* means that it already can be used for call-by-name application in a way: given a delayed argument $\mathrm{clos}\ \{\dots\} : \mathrm{Clos}(\mathrm{Eval}(\mathrm{Ret}(\mathrm{Val\,Int})))$ that will eventually return an integer, it can be passed to *idRef* and it will be returned back unevaluated. However, as is painfully obvious from the type, there is a *lot* of costly indirection to this calling convention: after the caller passes the delayed argument to *idRef*, it will wait for *idRef* to return a closure that the caller can then evaluate and then wait again for the real answer. Yikes!

It would be better to cut down on all the back and forth. Even lazy languages evaluate *id x* only when the result x is needed. So *id* might as well do the evaluation itself, like so:

$$\begin{aligned}
&idEval &&: \forall\ \mathrm{ty}\ a : \mathrm{Type\,sub}\ \mathbf{comp}\,.\ \mathrm{Val}(\mathrm{Clos}(\mathrm{Eval}\ a)) \to \mathrm{Eval}\ a\\
&idEval &&= \{\ \mathrm{ty}\ a \cdot \mathrm{val\,ref}\ x : \mathrm{Clos}(\mathrm{Eval}\ a) \cdot \mathrm{eval\,sub} \to \langle \mathrm{ref}\ x.\ \mathrm{call} \parallel \mathrm{eval\,sub}\rangle\ \}
\end{aligned}$$

Notice the different type of *idEval*: its parameter x is a closure around a subroutine of type $a : \mathrm{sub}\ \mathbf{comp}$ that can be evaluated directly with no extra input. For example, $a = \mathrm{Ret}(\mathrm{Val\,Int})$ means the closure returns an integer, and $a = \mathrm{Ret}(\mathrm{Val\,Int} \times \mathrm{Val\,Float})$ means it returns an unboxed pair.

Unboxed sum fusion. Next, let's consider some unboxed sum types to see how they behave when combined together or with other unboxed types. For example, to translate the boolean *and* function:

$$\mathit{and}\ \mathrm{True}\ x = x \qquad\qquad \mathit{and}\ \mathrm{False}\ x = \mathrm{False}$$

we can use the usual encoding of booleans as $\mathrm{Bool} = 1 + 1$, where $\mathrm{True} = 1, ()$ and $\mathrm{False} = \mathtt{0}, ()$. When we try to rewrite this definition of *and* in Call-By-Unboxed-Value, we cannot just name the second boolean parameter x, because x is not a pattern of $1 + 1$. Instead, it *must* elaborate the possible shapes that x might be and replace them for x on both sides,[3] like so:

$$\begin{array}{l}\mathit{and} : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Eval}(\mathrm{Ret}\,\mathrm{Bool}) \\ \mathit{and} = \{1, () \cdot 1, () \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ 1, (); \qquad \mathtt{0}, () \cdot 1, () \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); \\ \qquad\quad 1, () \cdot \mathtt{0}, () \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); \qquad \mathtt{0}, () \cdot \mathtt{0}, () \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); \}\end{array}$$

Of course, there are four possible options, enumerated by the four different stack shapes that contain no atomic values. We might ask how this information might be represented in a real machine, and what other types might have the same run-time representation. For example, it's correct to expect that rewriting *and* with the type $(\mathrm{Bool} \times \mathrm{Bool}) \to \mathrm{Eval}(\mathrm{Ret}\,\mathrm{Bool})$ rearranges the parentheses slightly, but essentially corresponds to the same low-level code. What may be more surprising is that merging the two booleans together into another sum type $\mathrm{Bool} + \mathrm{Bool}$, or even folding the choice of function arguments into one big product has essentially no change at run-time. Here are two other versions of *and* (where we use the shorthand $\uparrow P = \mathrm{Eval}(\mathrm{Ret}\,P)$ from fig. 1):

$$\begin{array}{ll}\mathit{and}' : (\mathrm{Bool} + \mathrm{Bool}) \to \uparrow \mathrm{Bool} & \mathit{and}'' : (\uparrow \mathrm{Bool})\ \&\ (\uparrow \mathrm{Bool})\ \&\ (\uparrow \mathrm{Bool})\ \&\ (\uparrow \mathrm{Bool}) \\ \mathit{and}' = \{1, 1, () \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ 1, (); & \mathit{and}'' = \{1 \cdot 1 \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ 1, (); \\ \quad 1, \mathtt{0}, () \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); & \quad\quad 1 \cdot \mathtt{0} \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); \\ \quad \mathtt{0}, 1, () \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); & \quad\quad \mathtt{0} \cdot 1 \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); \\ \quad \mathtt{0}, \mathtt{0}, () \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); \} & \quad\quad \mathtt{0} \cdot \mathtt{0} \cdot \mathrm{eval\,sub} \to \mathbf{ret}\ \mathtt{0}, (); \}\end{array}$$

All three versions of *and* have equivalent run-time calling conventions, and are implemented in the exact same way: a single switch statement over the four possible options.

Of course, this implementation won't do if we want to interpret *and* non-strictly; we should take care that the second argument is never evaluated if it isn't needed in the answer. This can be done by passing delayed boolean-generating closures of type $\downarrow\uparrow \mathrm{Bool}$ (where we use the shorthand $\downarrow Q = \mathrm{Val}(\mathrm{Clos}\,Q)$ from fig. 1) as is usual in similar mixed evaluation order calculi [31, 56], like so:

$$\begin{array}{l}\mathit{andCBN} : \downarrow\uparrow \mathrm{Bool} \to \downarrow\uparrow \mathrm{Bool} \to \uparrow \mathrm{Bool} \\ \mathit{andCBN} = \{\mathrm{val\,ref}\ y : \mathrm{Clos} \uparrow \mathrm{Bool} \cdot \mathrm{val\,ref}\ x : \mathrm{Clos} \uparrow \mathrm{Bool} \cdot \mathrm{eval\,sub} \to \\ \qquad \mathbf{do}\ y.\,\mathrm{call}\,.\,\mathrm{eval\,sub}\ \mathbf{as}\ \{1, () \to \mathrm{ref}\ x.\,\mathrm{call}\,.\,\mathrm{eval\,sub}; \\ \qquad\qquad\qquad\qquad \mathtt{0}, () \to \mathbf{ret}\ \mathtt{0}, (); \}\end{array}$$

where we now start using familiar functional-style application from fig. 4. Here, the different boolean options can't be fused: they haven't been evaluated yet, and pattern-matching *must* stop at closures.

This fusion into a single complex (co)pattern is not a special for simple enumerations. *Any* unboxed sum containing *any* amount of atomic values are all fused into a single shape. For example,

$$\mathit{maybeAdd}\ \mathrm{Nothing}\ y = y \qquad\qquad \mathit{maybeAdd}\ (\mathrm{Just}\ x)\ y = x + y$$

[3] Although we can alleviate much of this burden through some additional syntactic sugar. See appendix A for how to do so.

is translated into Call-By-Unboxed-Value as (using the primitive $add : \mathrm{Val}\,\mathrm{Nat} \to \mathrm{Val}\,\mathrm{Nat} \to \uparrow \mathrm{Nat}$):

$$\begin{array}{l} maybeAdd : (1 + \mathrm{Val}\,\mathrm{Nat}) \to \mathrm{Val}\,\mathrm{Nat} \to \uparrow \mathrm{Nat} \\ maybeAdd = \{(\mathtt{0}, ()) \cdot \mathrm{val\,int}\, y \cdot \mathrm{eval\,sub} \quad \to \mathbf{ret}\, \mathrm{val\,int}\, y; \\ \qquad (1, \mathrm{val\,int}\, x) \cdot \mathrm{val\,int}\, y \cdot \mathrm{eval\,sub} \to \lambda add\ (\mathrm{val\,int}\, x)\ (\mathrm{val\,int}\, y).\ \mathrm{eval\,sub};\} \end{array}$$

Now, regrouping the parentheses will change the type of *maybeAdd*, but does not affect where information is stored or how it will be moved around. That means that the second argument **val** int y might as well be part of the first argument, as

$$\begin{array}{l} maybeAdd' : \mathrm{Val}\,\mathrm{Nat} + (\mathrm{Val}\,\mathrm{Nat} \times \mathrm{Val}\,\mathrm{Nat}) \to \uparrow \mathrm{Nat} \\ maybeAdd' = \{(\mathtt{0}, \mathrm{val\,int}\, y) \cdot \mathrm{eval\,sub} \quad \to \mathbf{ret}\, \mathrm{val\,int}\, y; \\ \qquad (1, (\mathrm{val\,int}\, x, \mathrm{val\,int}\, y)) \cdot \mathrm{eval\,sub} \to \lambda add\ (\mathrm{val\,int}\, x)\ (\mathrm{val\,int}\, y).\ \mathrm{eval\,sub};\} \end{array}$$

or the one function can be divided into a product of two as

$$\begin{array}{l} maybeAdd'' : (\mathrm{Val}\,\mathrm{Nat} \to \uparrow \mathrm{Nat}) \,\&\, (\mathrm{Val}\,\mathrm{Nat} \to \mathrm{Val}\,\mathrm{Nat} \to \uparrow \mathrm{Nat}) \\ maybeAdd'' = \{\mathtt{0} \cdot \mathrm{val\,int}\, y \cdot \mathrm{eval\,sub} \quad \to \mathbf{ret}\, \mathrm{val\,int}\, y \\ \qquad 1 \cdot \mathrm{val\,int}\, x \cdot \mathrm{val\,int}\, y \cdot \mathrm{eval\,sub}; \to \lambda add\ (\mathrm{val\,int}\, x)\ (\mathrm{val\,int}\, y).\ \mathrm{eval\,sub};\} \end{array}$$

All three *maybeAdd* functions correspond to equivalent run-time code: a binary switch that loads one or two numbers into registers before deciding whether to add or not.

If we were unhappy with fusing the two arguments, we could forcibly separate them by boxing the first one, as $maybeAdd : \mathrm{Val}(\mathrm{Box}(1{+}\mathrm{Val}\,\mathrm{Nat})) \to \mathrm{Val}\,\mathrm{Nat} \to \uparrow \mathrm{Nat}$; this passes the first argument in a box, but otherwise it has the same evaluation order (both arguments must still be computed before *maybeAdd* is called). The non-strict version, of type $maybeAdd : \downarrow\uparrow(1{+}\mathrm{Val}\,\mathrm{Nat}) \to \downarrow\uparrow \mathrm{Nat} \to \uparrow \mathrm{Nat}$, naturally has its arguments separated into two heap-allocated closures.

Higher-order calling conventions. Now, we'll see how the four different kinds of types give greater precision over calling conventions for higher-order functions. The most basic one, $app\ f\ x = f\ x$, translated to Call-By-Unboxed-Value becomes:

$$\begin{array}{l} app : \forall\, \mathrm{ty}\, a : \mathrm{Type\ ref}\ \mathbf{val}\, .\forall\, \mathrm{ty}\, b : \mathrm{Type\ sub}\ \mathbf{comp}\, .\, \downarrow(\mathrm{Val}\, a \to \mathrm{Eval}\, b) \to \mathrm{Val}\, a \to \mathrm{Eval}\, b \\ app = \{\mathrm{ty}\, a \cdot \mathrm{ty}\, b \cdot \mathrm{val\,ref}\, f \cdot \mathrm{val\,ref}\, x \cdot \mathrm{eval\,sub} \to \langle f.\, \mathrm{call} \parallel \mathrm{val\,ref}\, x \cdot \mathrm{eval\,sub}\rangle\} \end{array}$$

Here, we must pass the function and its argument to *app* so we need to know their representations to even write the function code: a closure f is always a reference, but $x : a$ might be anything, so we pick $a : \mathrm{Type\ ref}\ \mathbf{val}$ to specify it is a reference, too. We need to know how to call f, so we assume that f can be evaluated as a sub-routine after being given exactly one argument (a reference); this requires b to be an atomic sub **comp**. Even still, we have the freedom to instantiate b to $\mathrm{Ret}(\mathrm{Val\,Int})$ to return just one result or $\mathrm{Ret}(\mathrm{Val\,Float} \times \mathrm{Val\,Nat} \times \mathrm{Val\,Clos}\, Q)$ to return an unboxed triple; that complex representation is irrelevant to *app*'s code. But maybe we can be even more generic. Recall that *app* can be η-reduced to $app'\ f = f$, which seems not to manipulate the second argument at all. In fact, this definition is the same as the identity function, which we can reuse as

$$\begin{array}{l} app' : \forall a : \mathrm{Type}\ \mathbf{cplx\ val}\, .\forall b : \mathrm{Type}\ \mathbf{cplx\ comp}\, .\, \downarrow(a \to b) \to \uparrow(\mathrm{Clos}(a \to b)) \\ app' = \{\mathrm{ty}\, a \cdot \mathrm{ty}\, b \cdot \mathrm{val}\, f : \mathrm{Clos}(a \to b) \cdot \mathrm{eval\,sub} \to \lambda idRef\ (\mathrm{Clos}(a \to b))\ f.\, \mathrm{eval\,sub}\} \end{array}$$

This time, a and b can be any complex types; they are never relevant to (co)pattern matching.

If we want to pass more than one argument at a time in a higher-order call, like $dup\ f\ x = f\ x\ x$, it can be translated to Call-By-Unboxed-Value as

$$\begin{array}{l} dup : \forall\, \mathrm{ty}\, a : \mathrm{Type\ ref}\ \mathbf{val}\, .\forall\, \mathrm{ty}\, b : \mathrm{Type\ sub}\ \mathbf{comp}\, .\, \downarrow(\mathrm{Val}\, a \to \mathrm{Val}\, a \to \mathrm{Eval}\, b) \to \mathrm{Val}\, a \to \mathrm{Eval}\, b \\ dup = \{\mathrm{ty}\, a \cdot \mathrm{ty}\, b \cdot \mathrm{val\,ref}\, f \cdot \mathrm{val\,ref}\, x \cdot \mathrm{eval\,sub} \to \langle f.\, \mathrm{call} \parallel \mathrm{val\,ref}\, x \cdot \mathrm{val\,ref}\, x \cdot \mathrm{eval\,sub}\rangle\} \end{array}$$

Here, we can assume that f. call is expecting *exactly* two (reference) arguments, which are being passed in the same call—anything else would be a type error because the call stack val ref $x \cdot$ val ref $x \cdot$ eval sub cannot match a copattern naming only one argument or naming three arguments—so *dup*'s code only has to handle the case of a perfect arity match, like [15].

Dictionary-passing type classes. One of the more exciting applications of [19] is generalizing over unboxed representations used for type classes. For example, a simplified numeric type class

$$\begin{aligned} \textbf{class}\,\mathrm{Num}\,a\,\textbf{where}\;\; (+) &\;\;::a \to a \to a \\ \mathit{negate} &::a \to a \end{aligned}$$

introduces overloaded operators $(+) :: \mathrm{Num}\,a \Rightarrow a \to a \to a$ and $\mathit{negate} :: \mathrm{Num}\,a \Rightarrow a \to a \to a$ that work for any instance of Num a. Ideally, we would like to have efficient instances of Num for various kinds of unboxed numeric types like Int and Float, but that's only possible if these are valid specializations of a. Eisenberg and Peyton Jones [19] allow for this through the use of polymorphism over representations. The Call-By-Unboxed-Value λ-calculus that we've introduced here only has monomorphic representations; nevertheless, it can still express the same generalization because *all* unboxed types have the same kind **cplx val** with no further specificity.

To see how the unspecified **cplx val** helps, consider how type classes are typically compiled using dictionary-passing style. The type class declaration of Num introduces a type of Num dictionaries—tuples of closures implementing each operation—that we would translate as:

$$\mathrm{Num}(\mathrm{ty}\,a : \textbf{cplx val}) : \textbf{cplx val} = \mathrm{Clos}(a \to a \to \uparrow a) \times \mathrm{Clos}(a \to \uparrow a)$$

The Num $a \Rightarrow \ldots$ constraint in generic code—like $(+)$ and *negate* themselves, or other functions defined in terms of them—is then translated as a regular parameter of the dictionary type Num a that the code uses to extract concrete definitions of the Num a operations. There is clearly no hope for defining overloaded operators of type $\mathit{negate} : \forall\,\mathrm{ty}\,a : \mathrm{Type}\,\textbf{cplx val}.\ \mathrm{Num}\,a \to a \to \uparrow a$ because there is no pattern for an unknown ty a : **cplx val**. However, we *can* easily implement these functions which merely extract and return one of the closures in the dictionary.

$(+) : \forall\,\mathrm{ty}\,a : \mathrm{Type}\,\textbf{cplx val}.\ \mathrm{Num}\,a \to \uparrow\downarrow(a \to a \to \uparrow a)$
$(+) = \{\,\mathrm{ty}\,a \cdot (\mathrm{val\,ref}\,f : \mathrm{Clos}(a \to a \to \uparrow a), \mathrm{val\,ref}\,g : \mathrm{Clos}(a \to \uparrow a)) \cdot \mathrm{eval\,sub} \to \textbf{ret}\,\mathrm{val\,ref}\,f\,\}$
$\mathit{negate} : \forall\,\mathrm{ty}\,a : \mathrm{Type}\,\textbf{cplx val}.\ \mathrm{Num}\,a \to \uparrow\downarrow(a \to \uparrow a)$
$\mathit{negate} = \{\,\mathrm{ty}\,a \cdot (\mathrm{val\,ref}\,f : \mathrm{Clos}(a \to a \to \uparrow a), \mathrm{val\,ref}\,g : \mathrm{Clos}(a \to \uparrow a)) \cdot \mathrm{eval\,sub} \to \textbf{ret}\,\mathrm{val\,ref}\,g\,\}$

Notice how the subtle—but essential!—detail that a *closure* is returned, as opposed to these operations calculating the result themselves, is recorded very conspicuously in the $\uparrow\downarrow$ shift in the types. Later, specific instances of Num a are just values of the dictionary type Num a. When picking a, they may choose types with any representation at all; since the instance chooses the a, it also knows how it is represented. For example, the unboxed integer and floating-point instances for Num are:

$\mathit{NumInt} : \mathrm{Num\,Int}$
$\mathit{NumInt} = (\mathrm{clos}\ \mathit{add\#}, \mathrm{clos}\ \mathit{negate\#})$

$\mathit{NumFlt} : \mathrm{Num\,Float}$
$\mathit{NumFlt} = (\mathrm{clos}\ \mathit{addFlt\#}, \mathrm{clos}\ \mathit{negateFlt\#})$

5 TRANSLATING FUNCTIONAL PROGRAMS TO CALL-BY-UNBOXED-VALUE

To be sure that unboxed data structures and call stacks don't cause any unintended issues, Call-By-Unboxed-Value should faithfully preserve the semantics of source-level functional programs that don't mention unboxed types at all. Rather than studying strict and non-strict languages separately, we will just demonstrate how to embed Call-By-Push-Value, since it subsumes both. And since polymorphism is one of our primary concerns, we extend Call-By-Push-Value with polymorphism

Translation of types $CBPV[\![A]\!] = A'$ and $CBPV[\![\underline{A}]\!] = B'$ and kinds $CBPV[\![\tau]\!] = \tau'$

$$
\begin{aligned}
CBPV[\![\mathbf{val}]\!] &= \text{ref } \mathbf{val} & CBPV[\![\mathbf{comp}]\!] &= \text{sub } \mathbf{comp} \\
CBPV[\![1]\!] &= \text{Box } 1 & CBPV[\![X]\!] &= X \\
CBPV[\![A_0 \times A_1]\!] &= \text{Box}(\text{Val } CBPV[\![A_0]\!] \times \text{Val } CBPV[\![A_1]\!]) & CBPV[\![A \to \underline{B}]\!] &= \text{Proc}(\text{Val } CBPV[\![A]\!] \to \text{Eval } CBPV[\![\underline{B}]\!]) \\
CBPV[\![0]\!] &= \text{Box } 0 & CBPV[\![\top]\!] &= \text{Proc } \top \\
CBPV[\![A_0 + A_1]\!] &= \text{Box}(\text{Val } CBPV[\![A_0]\!] + \text{Val } CBPV[\![A_1]\!]) & CBPV[\![\underline{B}_0 \,\&\, \underline{B}_1]\!] &= \text{Proc}(\text{Eval } CBPV[\![\underline{B}_0]\!] \,\&\, \text{Eval } CBPV[\![\underline{B}_1]\!]) \\
CBPV[\![\exists X{:}\tau.A]\!] &= \text{Box}(\exists\, \text{ty } X{:}\,\text{Type } CBPV[\![\tau]\!].CBPV[\![A]\!]) & CBPV[\![\forall X{:}\tau.\underline{B}]\!] &= \text{Proc}(\forall\, \text{ty } X{:}\,\text{Type } CBPV[\![\tau]\!].\ CBPV[\![\underline{B}]\!]) \\
CBPV[\![\text{U } \underline{B}]\!] &= \text{Clos}(\text{Eval } CBPV[\![\underline{B}]\!]) & CBPV[\![\text{F } A]\!] &= \text{Ret}(\text{Val } CBPV[\![A]\!])
\end{aligned}
$$

Translation of values $CBPV[\![V]\!] = V'$

$$
\begin{aligned}
CBPV[\![x]\!] &= \text{ref } x & CBPV[\![\text{thunk } M]\!] &= \text{clos } \{\, \text{eval sub} \to CBPV[\![M]\!] \,\} \\
CBPV[\![()]\!] &= \text{box}() & CBPV[\![(V, V')]\!] &= \text{box}(\text{val } CBPV[\![V]\!], \text{val } CBPV[\![V']\!]) \\
CBPV[\![(b, V)]\!] &= \text{box}(b, \text{val } CBPV[\![V]\!]) & CBPV[\![(T, V)]\!] &= \text{box}(CBPV[\![T]\!], \text{val } CBPV[\![V]\!])
\end{aligned}
$$

Translation of computations $CBPV[\![M]\!] = M'$

$$
\begin{aligned}
CBPV[\![\mathbf{match}\ V\ \mathbf{as}\ \{\, p_i \to M_i \overset{i \in I}{\cdots} \,\}]\!] &= \mathbf{unbox}\ CBPV[\![V]\!]\ \mathbf{as}\ \{\, p_i[\text{val ref } x/x, \overset{x \in FV(p_i)}{\cdots}] \to CBPV[\![M_i]\!] \overset{i \in I}{\cdots} \,\} \\
CBPV[\![\mathbf{match}\ V\ \mathbf{as}\ \{\, (X, y) \to M \,\}]\!] &= \mathbf{unbox}\ CBPV[\![V]\!]\ \mathbf{as}\ \{\, (\text{ty } X, \text{val ref } y) \to CBPV[\![M]\!] \,\} \\
CBPV[\![\mathbf{do}\ x \leftarrow M\ \mathbf{in}\ M']\!] &= \mathbf{do}\ CBPV[\![M]\!]\ \mathbf{as}\ \{\, \text{val ref } x \to CBPV[\![M']\!] \,\} \\
CBPV[\![\mathbf{return}\ V]\!] &= \mathbf{ret}\ \text{val } CBPV[\![V]\!]
\end{aligned}
$$

$$
\begin{aligned}
CBPV[\![\lambda x.M]\!] &= \text{proc } \{\, \text{val ref } x \cdot \text{eval sub} \to CBPV[\![M]\!] \,\} \\
CBPV[\![M\ V]\!] &= \langle CBPV[\![M]\!].\,\text{enter} \parallel \text{val ref } CBPV[\![V]\!] \cdot \text{eval sub} \rangle \\
CBPV[\![\lambda\ \{\ \}]\!] &= \text{proc } \{\ \} \\
CBPV[\![\lambda\ \{\, b.M_b \overset{b \in \{0,1\}}{\cdots} \,\}]\!] &= \text{proc } \{\, b \cdot \text{eval sub} \to CBPV[\![M_b]\!] \overset{b \in \{0,1\}}{\cdots} \,\} \\
CBPV[\![M\ b]\!] &= \langle CBPV[\![M]\!].\,\text{enter} \parallel b \cdot \text{eval sub} \rangle \\
CBPV[\![\Lambda X.M]\!] &= \text{proc } \{\, \text{ty } X \cdot \text{eval sub} \to CBPV[\![M]\!] \,\} \\
CBPV[\![M\ T]\!] &= \langle CBPV[\![M]\!].\,\text{enter} \parallel CBPV[\![T]\!] \cdot \text{eval sub} \rangle \\
CBPV[\![V.\,\text{force}]\!] &= \langle CBPV[\![V]\!].\,\text{call} \parallel \text{eval sub} \rangle
\end{aligned}
$$

Fig. 10. The translation from Polymorphic Call-By-Push-Value to Call-By-Unboxed-Value.

à la System F—with universal abstraction $\Lambda X : \tau.M$ and application $M\ T$ and existential packages (T, V) and unpacking $\mathbf{match}\ V\ \mathbf{as}\ (X : \tau, x : A) \to M$—without mention of representations.[4]

We can then embed this Polymorphic Call-By-Push-Value λ-calculus into Call-By-Unboxed-Value as shown in fig. 10. The key idea of this embedding is to interpret the (potentially polymorphic) types with uniform representations. Every value type $A : \mathbf{val}$ is an atomic reference $[\![A]\!] : \text{ref } \mathbf{val}$, and every computation type $\underline{B} : \mathbf{comp}$ is an atomic subroutine $[\![\underline{B}]\!] : \text{sub } \mathbf{comp}$. Uniform representation, unsurprisingly, forces boxes around every complex value (tuples, sum types, and packages). On the computation side, all computation is coerced to simple subroutines via sub-procedures (proc $\{ \ldots \}$) that we can just evaluate. Note that the Proc Q type hasn't appeared much in the examples seen thus far (in section 4), but here they are absolutely essential: the semantics of proc preserves the extensional properties (*i.e.,* η and sequencing equalities) of Call-By-Push-Value computation types. Without Proc Q, we would be forced to return closures instead, which is *observably different* from the source semantics. As such, Call-By-Push-Value is equivalent to an aggressively boxed subset of Call-By-Unboxed-Value that preserves not just typing but also all program equalities.

[4]For the full formal definition, see appendix B.

Theorem 5.1 (Type Preservation). *Let $[\![\_]\!]$ denote CBPV$[\![_]\!]$ in the following:*

(1) $\Gamma \vdash A : \tau$ *in Polymorphic CBPV if and only if* $[\![\Gamma]\!] \vdash [\![A]\!] : [\![\tau]\!]$ *in CBUV.*
(2) $\Gamma \vdash V : A$ *in Polymorphic CBPV if and only if* $[\![\Gamma]\!] \vdash [\![V]\!] : [\![A]\!] : \mathrm{ref}\ \mathbf{val}$ *in CBUV.*
(3) $\Gamma \vdash M : \underline{A}$ *in Polymorphic CBPV if and only if* $[\![\Gamma]\!] \vdash [\![M]\!] : [\![\underline{B}]\!] : \mathrm{sub}\ \mathbf{comp}$ *in CBUV.*

Theorem 5.2 (Soundness & Completeness). *Polymorphic Call-By-Push-Value's equational theory is sound and complete with respect to Call-By-Unboxed-Value:* $M = M'$ *iff.* CBPV$[\![M]\!]$ = CBPV$[\![M']\!]$.

Optimizing Away Boxes and Closures. One might notice the translation in fig. 10 gives code with drastically more indirection—and thus worse performance—than the examples given in fig. 2 and section 4. How do we actually use call-by-unboxed-value in a compiler to express optimizations which avoid boxes and currying to generate efficient code? One temptation is to give a better compilation translation with less indirection, but this involves some non-trivial understanding the source to identify boxes and closures that can be safely eliminated without changing the results.

An alternative approach is the *worker/wrapper transformation* [21] used by the Glasgow Haskell Compiler and previous work [15, 19, 47]. The idea is to naïvely translate source terms, and then *afterward* apply optimizations directly to the call-by-unboxed-value code to eliminate indirection. Since this optimization will involve changing the type of the code, it is split into two parts: the "worker" that is efficiently executes the function at a new type, and the "wrapper" that just calls the worker and marshalls between the old and the new types. For example, the naïve translation $[\![quotRem]\!]$ (fig. 2) can be optimized as the following wrapper *quotRem* and worker *quotRem′*:

$$
\begin{array}{l}
quotRem : CBPV[\![\mathrm{Nat} \to \mathrm{Nat} \to \mathrm{F}(\mathrm{Nat} \times \mathrm{Nat})]\!] \\
quotRem = \mathrm{proc}\{\mathrm{val\,ref}\ x \cdot \mathrm{eval\,sub} \to \mathrm{proc}\{\mathrm{val\,ref}\ y \cdot \mathrm{eval\,sub} \to \\
\qquad\qquad \mathbf{unbox}\ \mathrm{val\,int}\ x' \leftarrow x;\ \mathbf{unbox}\ \mathrm{val\,int}\ y' \leftarrow x; \\
\qquad\qquad \mathbf{do}\quad (\mathrm{val\,int}\ q, \mathrm{val\,int}\ r) \leftarrow \lambda quotRem'\ (\mathrm{val}\ x')\ (\mathrm{val}\ y').\ \mathrm{eval\,sub}; \\
\qquad\qquad \mathbf{ret}\quad \mathrm{box}(\mathrm{val}(\mathrm{box}(\mathrm{val}\ q)), \mathrm{val}(\mathrm{box}(\mathrm{val}\ r)))\ \}\}
\end{array}
$$

$$
\begin{array}{l}
quotRem' : \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Eval}(\mathrm{Ret}(\mathrm{Val\,Nat} \times \mathrm{Val\,Nat})) \\
quotRem' = \{\ \mathrm{val\,int}\ x \cdot \mathrm{val\,int}\ y \cdot \mathrm{eval\,sub} \to \\
\qquad\qquad \mathbf{do}\quad \mathrm{val\,ref}\ z \leftarrow CBPV[\![quotRem]\!].\ \mathrm{enter}(\mathrm{val}(\mathrm{box}(\mathrm{val}\ x))).\ \mathrm{enter}(\mathrm{val}(\mathrm{box}(\mathrm{val}\ y))).\ \mathrm{eval\,sub}; \\
\qquad\qquad \mathbf{unbox}\ (\mathrm{val\,ref}\ z_1, \mathrm{val\,ref}\ z_2) \leftarrow z;\ \mathbf{unbox}\ \mathrm{val\,int}\ n_1 \leftarrow z_1;\ \mathbf{unbox}\ \mathrm{val\,int}\ n_2 \leftarrow z_2; \\
\qquad\qquad \mathbf{ret}\quad (\mathrm{val}\ n_1, \mathrm{val}\ n_2)\}
\end{array}
$$

The code for the worker *quotRem′* is generated by applying the simple translation $[\![quotRem]\!]$ in a context that actually uses it; though this seems inefficient, other standard optimizations (based on the equational theory in section 3.4) can reduce it to the efficient form shown in fig. 2. The remaining wrapper *quotRem* is small and can be inlined aggressively; if a call site actually passes unboxed arguments to *quotRem* then this will simplify to a fast direct call to *quotRem′*.

6 AN UNBOXED ABSTRACT MACHINE

Having studied Call-By-Unboxed-Value from the high-level—as a suitable target for semantics-preserving compilation of functional programs—we now consider it from a lower-level perspective to be sure that it can actually be implemented with the intended memory behavior on realistic machines. Specifically, the *only* objects in long-term storage are reference values box S and clos F, as well as contiguously-stored subroutine stack frames corresponding to $\mathbf{do}\ \square\ \mathbf{as}\ G$ and $\langle \square.\ \mathrm{enter}\ \|K\rangle$—everything else can be held in a simple but fast register locations. Moreover, our contribution is to show how programs can be compiled and run using *only* the information in their syntax, ignoring all typing information at compile-time and run-time, but nevertheless preserving typability.

6.1 Annotated machine code

Call-By-Unboxed-Value variables are already annotated with fixed representations and the evaluation of functions is annotated by what kind of computation to expect. The missing information to compile code is about closure environments: when code pointers are stored, we need to know what are the relevant free variables to copy into the closure, and thus how they are represented. We can explicate this information by extendind the Call-By-Unboxed-Value syntax like so

$$\mathit{Value} \ni V ::= \cdots \mid \mathrm{clos}\, F\ [\Gamma] \qquad\qquad \mathit{Comp} \ni M ::= \cdots \mid \mathbf{do}\, M\, \mathbf{as}\, G\ [\Gamma, O]$$

Thankfully, closure information is easy to recover just from the program itself—whether or not we have any type-checking information. In fact, we can even annotate ill-typed programs, though they may go wrong at run-time. The most interesting steps for compiling atomic values ($AM[\![V]\!]_\Gamma = V'$), computations ($AM[\![M]\!]^O_\Gamma = M'$), and (co)matching code ($AM[\![G]\!]^O_\Gamma = G'$ and $AM[\![F]\!]_\Gamma = F'$) are:

$$\begin{aligned}
AM[\![\mathrm{clos}\, F]\!]_\Gamma &= \mathrm{clos}\, AM[\![F]\!]_\Gamma[\Gamma|_{FV(F)}] \\
AM[\![\mathbf{do}\, M\, \mathbf{as}\, G]\!]^O_\Gamma &= \mathbf{do}\, AM[\![M]\!]^{\mathrm{sub}}_\Gamma\, \mathbf{as}\, AM[\![G]\!]^O_\Gamma[\Gamma|_{FV(G)}, O] \\
AM[\![\{\, k[\Gamma_k, O_k] \to M_k \overset{k\in Q}{\ldots} \,\}]\!]_\Gamma &= \{\, k[\Gamma_k, O_k] \to AM[\![M_k]\!]^{O_k}_{\Gamma_k,\Gamma} \overset{k\in Q}{\ldots} \,\} \\
AM[\![\{\, s[\Gamma_s] \to M_s \overset{s\in P}{\ldots} \,\}]\!]^O_\Gamma &= \{\, s[\Gamma_s] \to AM[\![M_s]\!]^O_{\Gamma_s,\Gamma} \overset{s\in P}{\ldots} \,\}
\end{aligned}$$

where the parameter Γ collects information about the local variables from their binding sites, and O is the expected observation of a computation. The operation $\Gamma|_{FV(F)}$ means to restrict Γ to only the free variables actually found in F (*i.e.,* $FV(F)$). As shorthand for inspecting copatterns, we write $k[\Gamma, O]$ to mean k ends in eval O. The rest of the cases follow directly by induction. Of note, we can always determine how to observe computation sub-terms from context. Usually, this O comes from the expectation imposed on matching code G (as in **do** above) or from a surrounding copattern, but in M. eval we know M should have some sort of Proc Q type, which is always a subroutine computation, thus $AM[\![M.\,\mathrm{enter}]\!]_\Gamma = AM[\![M]\!]^{\mathrm{sub}}_\Gamma$. eval.

6.2 Machine configurations and transitions

The abstract machine is defined in fig. 11. Notice that a machine configuration m combines three parts: a command c saying what to do, local registers ρ and κ, and long-term storage σ. Each ρ register is fixed to one atomic representation R and only holds compatible R-represented values W: numeric constants, reference pointers (ref x) into storage, or *closed* types T. As such, reading or writing a variable's value in ρ requires knowing its name *and* its representation. While type registers $[\mathrm{ty}\ x := T]$ may seem to hold a large, complex type T, the ty representation denotes a phantom register that is erased for real execution; it is only maintained hypothetically to correspond with typing information from the source language. κ denotes the context of evaluation, and points to the top of the call stack sub $\overline{x}$ during a subroutine computation, or is empty during a run computation.

Long-term storage σ contains a combination of heap objects $[x := H]$ as well as stack frames $[\overline{x} := E]$. The two address spaces are kept separate to accommodate distinct allocation strategies: x addresses are heap-allocated and garbage collected, but $\overline{x}$ addresses follow a linear stack discipline and can be allocated and freed as a traditional, contiguous call stack. Stored code objects, clos $F[\rho]$ and **do** $G[\rho\kappa]$, are closed over the contents of (value and stack) registers at storage time.

At times, we need to simplify a sequence of values $V\ldots$ into a sequence of constants $W\ldots$ that can actually be stored locally in registers. This is done through the multi-value storing operation $\rho^*(V\ldots)$ that returns both a sequence of constants $W\ldots$, and in doing so, may need to allocate some heap objects that come from $V\ldots$. For example, just storing $\rho^*(\mathrm{clos}\, F[\Gamma])$ will allocate the closure clos $F[\rho|_\Gamma]$ (where $\rho|_\Gamma$ denotes the restriction of ρ to only variables listed in Γ) to some location x on the heap, and return the pair ref x; $[x := \mathrm{clos}\, F[\rho|_\Gamma]]$. In the other direction, sometimes we

$$\begin{array}{rlrl}
\mathit{RegValue} \ni W & ::= n \mid n.n \mid T \mid \mathrm{ref}\ x & \mathit{Registers} \ni \rho & ::= \bullet \mid R\ x := W; \rho \\
\mathit{HeapObject} \ni H & ::= \mathrm{box}\ s[W\ldots] \mid \mathrm{clos}\ F[\rho] & \mathit{Store} \ni \sigma & ::= \bullet \mid \sigma, x := H \mid \overline{x} := E, \sigma \\
\mathit{StackFrame} \ni E & ::= \mathrm{enter}\ k[W\ldots\kappa] \mid \mathbf{do}\ G[\rho\kappa] & \mathit{StackReg} \ni \kappa & ::= \mathrm{run} \mid \mathrm{sub}\ \overline{x} \\
\mathit{Machine} \ni m & ::= c[\rho\kappa][\sigma] & \mathit{Command} \ni c & ::= \langle M \rangle \mid \langle s \parallel W\ldots \parallel G \rangle \mid \langle F \parallel k \parallel W\ldots \rangle
\end{array}$$

$$\frac{V\ldots; \rho' = \mathit{Load}_f(k, W\ldots) \quad f(k[V\ldots]) = \mathrm{terminal}}{\langle f \parallel k \parallel W\ldots\rangle\ [\rho\kappa][\sigma]\ \mathrm{terminal}} \qquad \frac{V\ldots; \rho' = \mathit{Load}_g(s, W\ldots) \quad g(s[V\ldots]) = \mathrm{terminal}}{\langle s \parallel W\ldots \parallel g\rangle\ [\rho\ \mathrm{run}][\sigma]\ \mathrm{terminal}}$$

Transition rules $\boxed{m \mapsto m'}$

$$\begin{array}{rl}
(\mathbf{do}) & \langle \mathbf{do}\ M\ \mathbf{as}\ G[\Gamma, O]\rangle\ [\rho\kappa][\sigma] \mapsto \langle M\rangle\ [\rho\ \mathrm{sub}\ \overline{x}][\overline{x} := \mathbf{do}\ G[\rho|_\Gamma\ \kappa(O)], \sigma] \\
(\mathrm{enter}) & \langle\langle M.\mathrm{enter} \parallel k[V\ldots, O]\rangle\rangle\ [\rho\kappa][\sigma] \mapsto \langle M\rangle\ [\rho\ \mathrm{sub}\ \overline{x}]\ [\overline{x} := \mathrm{enter}\ k[W\ldots, \kappa(O)], \sigma, \sigma'] \quad (\text{if } W\ldots; \sigma' = \rho^*(V\ldots)) \\
(\mathbf{as}) & \langle s[V\ldots]\ \mathbf{as}\ G\rangle\ [\rho\kappa][\sigma] \mapsto \langle s \parallel W\ldots \parallel G\rangle\ [\rho\kappa][\sigma, \sigma'] \quad (\text{if } W\ldots; \sigma' = \rho^*(V\ldots)) \\
(\lambda) & \langle\langle \lambda F \parallel k[V\ldots]\rangle\rangle\ [\rho\kappa][\sigma] \mapsto \langle F \parallel k \parallel W\ldots\rangle\ [\rho'\kappa][\sigma, \sigma'] \quad (\text{if } W\ldots; \sigma' = \rho^*(V\ldots)) \\
(\mathrm{Box}) & \langle \mathbf{unbox}\ V\ \mathbf{as}\ G\rangle\ [\rho\kappa][\sigma] \mapsto \langle s \parallel W\ldots \parallel G\rangle\ [\rho\kappa][\sigma, \sigma'] \quad (\text{if } \mathrm{box}\ s[W\ldots]; \sigma' = \sigma_\rho(V)) \\
(\mathrm{Clos}) & \langle\langle V'.\mathrm{call} \parallel k[V\ldots]\rangle\rangle\ [\rho\kappa][\sigma] \mapsto \langle F \parallel k \parallel W\ldots\rangle\ [\rho'\kappa][\sigma, \sigma'] \quad (\text{if } \mathrm{clos}\ F[\rho']; \bullet = \sigma_\rho(V') \text{ and } W\ldots; \sigma' = \rho^*(V\ldots)) \\
(\mathrm{Ret}) & \langle \mathbf{ret}\ s[V\ldots]\rangle [\rho\ \mathrm{sub}\ \overline{x}][\overline{x} := \mathbf{do}\ G[\rho'\kappa'], \sigma] \mapsto \langle s \parallel W\ldots \parallel G\rangle\ [\rho'\kappa'][\sigma, \sigma'] \quad (\text{if } W\ldots; \sigma' = \rho^*(V\ldots)) \\
(\mathrm{Proc}) & \langle \mathrm{proc}\ F\rangle\ [\rho\ \mathrm{sub}\ \overline{x}][\overline{x} := \mathrm{enter}\ k[W\ldots\kappa'], \sigma] \mapsto \langle F \parallel k \parallel W\ldots\rangle\ [\rho\kappa'][\sigma]
\end{array}$$

$$\begin{array}{rl}
(\mathit{Fun}) & \langle \{\, k[\Gamma_k] \to M_k \overset{k \in Q}{\ldots} \parallel k' \parallel W\ldots \,\}\rangle\ [\rho\kappa][\sigma] \mapsto \langle M_{k'}\rangle\ [\rho, (\Gamma_{k'} := W\ldots)\kappa][\sigma] \\
(\mathit{Match}) & \langle s' \parallel W\ldots \parallel \{\, s[\Gamma_s] \to M_s \overset{s \in P}{\ldots} \,\}\rangle\ [\rho\kappa][\sigma] \mapsto \langle M_{s'}\rangle\ [\rho, (\Gamma_{s'} := W\ldots)\kappa][\sigma] \\
(\mathit{Prim}_f) & \langle f \parallel k \parallel W\ldots\rangle\ [\rho\kappa][\sigma] \mapsto \langle \mathit{Prim}_f(k[V\ldots])\rangle\ [\rho'\kappa][\sigma] \quad (\text{if } V\ldots; \rho' = \mathit{Load}_f(k, W\ldots)) \\
(\mathit{Prim}_g) & \langle s \parallel W\ldots \parallel g\rangle\ [\rho\kappa][\sigma] \mapsto \langle \mathit{Prim}_g(s[V\ldots])\rangle\ [\rho'\kappa][\sigma] \quad (\text{if } V\ldots; \rho' = \mathit{Load}_g(s, W\ldots))
\end{array}$$

Value loading $\boxed{\sigma_\rho(V) = H; \sigma}$, value storing $\boxed{\rho^*(V\ldots) = W\ldots; \sigma}$, and primitive operations

$$\begin{array}{rl}
\sigma_\rho(\mathrm{clos}\ F[\Gamma]) = \mathrm{clos}\ F[\rho|_\Gamma]; \bullet & \sigma_\rho(\mathrm{ref}\ x) = \sigma(\rho(\mathrm{ref}\ x)); \bullet \\
\sigma_\rho(\mathrm{box}\ s[V\ldots]) = \mathrm{box}\ s[W\ldots]; \sigma' \quad (\text{if } W\ldots; \sigma' = \rho^*(V\ldots)) & \\
\rho^*(V\ldots, R\ x) = W\ldots, \rho(R\ x); \sigma & (\text{if } \rho^*(V\ldots) = W\ldots; \sigma) \\
\rho^*(V\ldots, \mathit{const}) = W\ldots, \mathit{const}; \sigma & (\text{if } \mathit{const} \in \{n, n.n\}, \rho^*(V\ldots) = W\ldots; \sigma) \\
\rho^*(V\ldots, \mathrm{box}\ s[V'\ldots]) = W\ldots, \mathrm{ref}\ x; (x := \mathrm{box}\ s[W'\ldots], \sigma', \sigma) & (\text{if } \rho^*(V'\ldots) = W'\ldots; \sigma', \rho^*(V\ldots) = W\ldots; \sigma) \\
\rho^*(V\ldots, \mathrm{clos}\ F[\Gamma]) = W\ldots, \mathrm{ref}\ y; (y := \mathrm{clos}\ F[\rho|_\Gamma], \sigma', \sigma) & (\text{if } \rho^*(V\ldots) = W\ldots; \sigma) \\
\rho^*(V\ldots, T) = W\ldots, T[\rho(\mathrm{ty}\ x)/\mathrm{ty}\ x \overset{x \in FV(T)}{\ldots}]; \sigma & (\text{if } \rho^*(V\ldots) = W\ldots; \sigma) \\
\mathit{Prim}_f(k[V\ldots, O]) = AM[\![f(k[V\ldots])]\!]^O_{\mathit{Sig}_f(k)} & \mathit{Prim}_g(g, s[V\ldots]) = AM[\![g(s[V\ldots])]\!]^{\mathrm{run}}_{\mathit{Sig}_g(s)}
\end{array}$$

Fig. 11. The Call-By-Unboxed-Value abstract machine.

need the real value of V from source code, which may be a variable reference. $\sigma_\rho(V)$ looks up V if it is a reference into σ, and returns the heap object representation in either case.

Lastly, the command c itself may take three different forms. $\langle M\rangle$ is the standard command, which just executes the given computation M corresponding to the operational semantics. The other two, $\langle s \parallel W\ldots \parallel G\rangle$ and $\langle F \parallel k \parallel W\ldots\rangle$ are intermediate states that unify the cases where a pattern or copattern match is ready to happen. When F or G are defined as source code $\{\, k[\Gamma] \to M\ldots \,\}$

or $\{\, s[\Gamma] \to M\ldots \,\}$, then the *Fun* and *Match* rules do a switch statement on the shape s or k, and then bind the associated values $W\ldots$ to the registers named by Γ. For primitive operations f and g, we assume that they are implemented in a way compatible with the operational semantics, as specified by $Prim_f$ and $Prim_g$, which fundamentally relies on the parametricity of references (assumption 3.1). Specifically, runtime-allocated store locations can't be compiled into the definitions of primitive operations. Thus, we also assume that each operation has an associated run-time parameter signature $Sig_f(k) = \Gamma$ and $Sig_g(s) = \Gamma$ known at compile-time for each shape, as well as a parameter-loading routine $Load_f(k, W\ldots) = V\ldots; \rho$ and $Load_g(s, W\ldots) = V\ldots; \rho$ that abstracts out (at least) all the in $W\ldots$ into ρ and replaces them with their ρ-bound names.

Aside 6.1. It may seem like matching on shapes k and s in the *Fun* and *Match* rules would be a complex operation to implement. But remember: shapes are devoid of any information about the atomic values held "inside," and don't even assign some name to positions, leaving them blank $\Box$s. Furthermore, structures and stacks are second-class and must be fully formed in-place, since the syntax statically separates shapes from their contents. The only run-time requirement of complex types of shapes is that they are all distinct. As such, the whole shape can be reduced to a single constant and choosing a branch is just one switch statement. Once the branch has been selected, the associated atomic values $W\ldots$ can be assigned their local names to evaluate the next computation.

For example, a pointer $x : \mathrm{Box}((\mathrm{Val\,Int} + \mathrm{Val\,Float} \times \mathrm{Val\,Int}) + 1)$ to a boxed value can have three possible shapes: (0) $0, 0, \mathrm{val}\ \Box$ contains a single int, (1) $0, 1, \mathrm{val}\ \Box, \mathrm{val}\ \Box$ contains a pair of a float and int, and (2) $1, ()$ contains nothing. To generate code, we need an enumeration mapping each shape to a different constant distinguish the options. For instance, we could use the numeric labels of the above enumeration, so $\mathrm{box}(0, 1, \mathrm{val}\ 3.14, \mathrm{val}\ 42)$ is represented as a pointer to a single-byte tag 1, a 64-bit floating-point 3.14, and a 32-bit integer 42. If x points to this sequence, the unboxing

$$\textbf{unbox}\ x\ \textbf{as}\ \{\ 0, 0, \mathrm{val\,int}\ y \to M_1; \qquad 0, 1, \mathrm{val\,flt}\ y, \mathrm{val\,int}\ z \to M_2; \qquad 1, () \to M_3\}$$

should be compiled to a C-like tagged union and switch statement as shown in fig. 12a.

Copatterns are more difficult to express in a C-like pseudo code, since we only know what arguments to expect *after* checking the tag denoting the stack frame's shape. However, it still follows the same principle in a lower-level language with an explicit call stack. All versions of the *maybeAdd* function should be compiled to the same low-level code with two possible stack shapes:

(0) $0 \cdot \mathrm{val}\ \Box \cdot \mathrm{eval\ sub}$ describes a frame with 1 integer argument and 1 return pointer, and
(1) $1 \cdot \mathrm{val}\ \Box \cdot \mathrm{val}\ \Box \cdot \mathrm{eval\ sub}$ describes a frame with 2 integer arguments and 1 return pointer.

As before, we can generate code for this complex type by following the enumeration to assign a numeric index to each stack shape. The function can be called by passing the number stack shape tag in the first register, followed by any additional parameters. fig. 12b shows an example of an Intel x86 implementation of register-passing code, where the tag is passed in `%al` and the arguments are passed in the remaining registers and spill onto the stack as usual. On the other side, *maybeAdd*'s code first starts with a switch or conditional to check the tag describing the stack shape, then jumps to the code for that case that knows how to access the remaining parameters for that branch.

6.3 Back-translation and bisimulation

The abstract machine in fig. 11 is meant to correctly implement the low-level details left abstract in operational semantics in fig. 5. To be sure that the two give the same results, we can show that the steps of the two systems remain in sync by relating machine configurations back to the source calculus. To decompile machine code ($AM[\![M]\!]^{-1}$, $AM[\![V]\!]^{-1}$, *etc.*), we just need to erase the extra annotations that were added to function closures and **do**-statements.

```c
struct {
  char tag;
  union {
    int zero; // case 0 = 0, 0, val int
    // case 1 = 0, 1, val flt, val int
    struct { float fst; int snd; } one;
    // empty case 2 = 1, ()
  } body;
} *x;
switch (x->tag) {
  case 0:
    int y = x->body.zero;
    M1...
    break;
  case 1:
    float y = x->body.one.fst;
    int z = x->body.one.snd;
    M2...
    break;
  case 2:
    M3...
}
```

(a) Unboxing a complex value in C.

```asm
# at the call site of: maybeAdd (1, val 10) (val 20).eval sub
  movb $1,  %al   # case 1 = (1, val []) · val [] · eval sub
  movl $10, %edi  # first argument = 10
  movl $20, %esi  # second argument = 20
  call maybeAdd   # maybeAdd (1, val 10) (val 20).eval sub

# at the function definition site...
maybeAdd:
  cmpb $1, %al    # check for case 1
  je   maybeAdd1  # jump to convention = 1 · val [] · val [] · eval sub
                  # otherwise, use convention = 0 · val [] · eval sub
maybeAdd0:
  # %edi holds the only argument y
  movl %edi, %eax  # only return result is y held in %eax
  ret              # return val int y

maybeAdd1:
  # %edi holds first argument, %esi holds second argument
  movl %edi, %eax
  addl %esi, %eax  # add both arguments
  ret              # return the only result
```

(b) Function code with complex calling conventions in x86 assembly.

Fig. 12. Examples of generating low-level code for pattern and copattern matching.

$$\text{Commands } \boxed{AM[\![c]\!]^{-1} = M} \text{ and configurations } \boxed{AM[\![m]\!]^{-1} = M}$$

$$AM[\![\langle M \rangle]\!]^{-1} = AM[\![M]\!]^{-1} \qquad AM[\![\langle s \parallel W... \parallel G \rangle]\!]^{-1} = s[W...] \text{ as } AM[\![G]\!]^{-1}$$

$$AM[\![c[\rho\kappa]]\!]^{-1}_{\sigma} = AM[\![\kappa]\!]^{-1}_{\sigma}[AM[\![c]\!]^{-1}[AM[\![\rho]\!]^{-1}]] \qquad AM[\![\langle F \parallel k \parallel W...O \rangle]\!]^{-1} = \langle \lambda AM[\![F]\!]^{-1} \parallel k[W...] \rangle$$

$$AM[\![c[\rho\kappa][\sigma]]\!]^{-1} = AM[\![c[\rho\kappa]]\!]^{-1}_{\sigma}[AM[\![\sigma]\!]^{-1}]$$

$$\text{Stack pointers } \boxed{AM[\![\kappa]\!]^{-1}_{\sigma} = E} \text{ and stored stack frames } \boxed{AM[\![E]\!]^{-1}_{\sigma} = E}$$

$$AM[\![\mathbf{do}\, G[\rho\kappa]]\!]^{-1}_{\sigma} = AM[\![\kappa]\!]^{-1}_{\sigma}\left[\mathbf{do}\, \square \,\mathbf{as}\, AM[\![G]\!]^{-1}[AM[\![\rho]\!]^{-1}]\right] \qquad AM[\![\text{run}]\!]^{-1}_{\sigma} = \square$$

$$AM[\![\text{enter}\, k[W...\kappa]]\!]^{-1}_{\sigma} = AM[\![\kappa]\!]^{-1}_{\sigma}\,[\langle \square.\, \text{enter} \parallel k[W...] \rangle] \qquad AM[\![\text{sub}\, \overline{x}]\!]^{-1}_{\sigma} = AM[\![\sigma(\overline{x})]\!]^{-1}_{\sigma}$$

$$\text{Registers } \boxed{AM[\![\rho]\!]^{-1} = W/R\, x...}, \text{ heap objects } \boxed{AM[\![H]\!]^{-1}_{\sigma} = V} \text{ and the heap } \boxed{AM[\![\sigma]\!]^{-1} = V/x...}$$

$$AM[\![\text{clos}\, F[\rho]]\!]^{-1} = \text{clos}\, AM[\![F]\!]^{-1}[AM[\![\rho]\!]^{-1}] \qquad AM[\![\text{box}\, s[W...]]\!]^{-1} = \text{box}\, s[W...]$$

$$AM[\![\overline{x} := E...]\!]^{-1} = \bullet \qquad AM[\![R\, x := W...]\!]^{-1} = W/R\, x...$$

$$AM[\![x := H, \sigma]\!]^{-1} = (AM[\![H]\!]^{-1}[AM[\![\sigma]\!]^{-1}])/x, AM[\![\sigma]\!]^{-1} \qquad (\text{if } x \notin FV(\sigma))$$

Fig. 13. Decompilation of the abstract machine.

Decompiling commands and configurations, as shown in fig. 13, takes more work. The main idea is that the registers (ρ) and the store (σ) hold information about deferred substitutions that would have happened already in the operational semantics. In $AM[\![c[\rho\kappa]]\!]^{-1}$, registers ρ are decompiled as a substitution applied to c, while the stack register κ gets rebuilt as an evaluation context surrounding c. The last step is to sort through the heap in σ and substitute each reconstructed heap object back into the computation. Doing so relies on the fact that σ is non-cyclic, which means we can always find (at least) one object that nothing else depends on to build. With this decompilation complete, we can create a bisimulation that links both semantics, under the assumption that primitive parameter passing correctly abstracts out values into registers.

Definition 6.2 (Bisimulation Relation). The bisimulation relation $M \sim m$ between closed Call-By-Unboxed-Value terms and closed configurations of the abstract machine is: $M \sim m$ iff $M = AM[\![m]\!]^{-1}$

Updated typing rules for annotated closures and do-sequences:

$$\frac{\Gamma|_\Delta \vdash F : Q\ ;}{\Gamma \vdash \mathrm{clos}\, F[\Delta] : \mathrm{Clos}\, Q : \mathrm{ref}\ \mathbf{val}}\ \mathrm{Clos}\, I \qquad \frac{\Gamma \vdash M : \mathbf{ret}\, P : \mathrm{sub}\ \mathbf{comp} \quad \Gamma|_\Delta\ ; G : P \vdash B : O\ \mathbf{comp}}{\Gamma \vdash \mathbf{do}\, M\, \mathbf{as}\, G[\Delta, O] : B : O\ \mathbf{comp}}\ \mathrm{Ret}\, E$$

$$\mathit{StoreEnv} \ni \Psi ::= \bullet \mid \Psi, x : A \qquad \mathit{StackEnv} \ni \Xi ::= \bullet \mid \overline{x} : B$$

Types for register values $\boxed{\Psi \vdash W : A : R\ \mathbf{val}}$ and the stack registers $\boxed{\kappa : \Phi \vdash \Xi}$

$$\frac{}{\Psi \vdash n : \mathrm{Int} : \mathrm{int}\ \mathbf{val}} \qquad \frac{}{\Psi \vdash n.n : \mathrm{Float} : \mathrm{flt}\ \mathbf{val}} \qquad \frac{}{\Psi, x : A \vdash x : A : \mathrm{ref}\ \mathbf{val}}$$

$$\frac{\bullet \vdash T : \tau}{\Psi \vdash T : \mathrm{Type}\, \tau : \mathrm{ty}\ \mathbf{val}} \qquad \frac{}{\mathrm{sub}\, \overline{x} : B : \mathrm{sub}\ \mathbf{comp} \vdash \overline{x} : B} \qquad \frac{}{\mathrm{run} : \mathrm{void} : \mathrm{run}\ \mathbf{comp} \vdash \bullet}$$

Types for heap objects $\boxed{\Psi \vdash H : A}$ and stack frames $\boxed{\Psi \mid E : B \vdash \Xi}$

$$\frac{\bullet \mid \Delta \vdash s : P\ ; \quad \Psi \vdash W\ldots : \Delta}{\Psi \vdash \mathrm{box}\, s[W\ldots] : \mathrm{Box}\, P} \qquad \frac{\Psi \vdash \rho : \Gamma \quad \Gamma \vdash F : Q\ ;}{\Psi \vdash \mathrm{clos}\, F[\rho] : \mathrm{Clos}\, Q}$$

$$\frac{\bullet \mid \Delta\ ; k : Q \vdash \Phi \quad \Psi \vdash W\ldots : \Delta \quad \Psi \mid \kappa : \Phi \vdash \Xi}{\Psi \mid \mathrm{enter}\, k[W\ldots\kappa] : \mathrm{Proc}\, Q \vdash \Xi} \qquad \frac{\Gamma\ ; G : P \vdash \Phi \quad \Psi \vdash \rho : \Gamma \quad \Psi \mid \kappa : \Phi \vdash \Xi}{\Psi \mid \mathbf{do}\, G[\rho\kappa] : \mathrm{Ret}\, P \vdash \Xi}$$

Typed value registers $\boxed{\Psi \vdash \rho : \Gamma}$ and sequences $\boxed{\Psi \vdash W\ldots : \Delta}$

$$\frac{\Psi \vdash W : A : R\ \mathbf{val} \quad \Psi \vdash \rho : \Gamma[W/R\, x]}{\Psi \vdash R\, x := W, \rho : (R\, x : A, \Gamma)} \qquad \frac{}{\Psi \vdash \bullet : \bullet} \qquad \frac{\Psi \vdash W : A : R\ \mathbf{val} \quad \Psi \vdash W'\ldots : \Delta[W/R\, x]}{\Psi \vdash W, W'\ldots : (R\, x : A, \Delta)}$$

Types for the long-term store $\boxed{\sigma : (\Psi \dashv \Xi)}$ binding heap objects (Ψ) and a top stack frame (Ξ)

$$\frac{}{\bullet : (\bullet \vdash \bullet)} \qquad \frac{\sigma : (\Psi \dashv \Xi) \quad \Psi \vdash H : A}{(\sigma, x := H) : (\Psi, x : A \dashv \Xi)} \qquad \frac{\Psi \mid E : B \vdash \Xi \quad \sigma : (\Psi \dashv \Xi)}{(\overline{x} := E, \sigma) : (\Psi \dashv \overline{x} : B)}$$

Machine commands $\boxed{\Psi \mid \Gamma \vdash c : \Phi}$ and closing configurations:

$$\frac{\Gamma \vdash M : \Phi}{\Psi \mid \Gamma \vdash \langle M \rangle : \Phi}$$

$$\frac{\Gamma \mid \Delta \vdash s : P\ ; \quad \Psi \vdash W\ldots : \Delta \quad \Gamma\ ; G : P \vdash \Phi}{\Psi \mid \Gamma \vdash \langle s \parallel W\ldots \parallel G \rangle : \Phi} \qquad \frac{\Gamma \vdash F : Q\ ; \quad \Gamma \mid \Delta\ ; k : Q \vdash \Phi \quad \Psi \vdash W\ldots : \Delta}{\Psi \mid \Gamma \vdash \langle F \parallel k \parallel W\ldots \rangle : \Phi}$$

$$\frac{\Psi \vdash \rho : \Gamma \quad \Psi \mid \Gamma \vdash c : \Phi \quad \kappa : \Phi \vdash \Xi}{c[\rho\kappa] : \Psi \vdash \Xi}\ \mathit{RegCut} \qquad \frac{c[\rho\kappa] : (\Psi \vdash \Xi) \quad \sigma : (\Psi \dashv \Xi)}{c[\rho\kappa][\sigma]\ \mathrm{OK}}\ \mathit{StoreCut}$$

Fig. 14. The Call-By-Unboxed-Value abstract machine type system.

Assumption 6.3. $V[AM[\![\rho]\!]^{-1}]\ldots = W\ldots$ for all $V\ldots; \rho = \mathit{Load}_f(k, W\ldots)$ or $V\ldots; \rho = \mathit{Load}_g(s, W\ldots)$.

Lemma 6.4 (Bisimilarity). *CBUV's operational semantics and abstract machine are bisimilar,*

(1) For all closed M, $M \sim \langle AM[\![M]\!]^{\mathrm{run}}_{\bullet} \rangle\ [\mathrm{run}]\, [\bullet]$,
(2) for all closed m with a non-cyclic σ, $AM[\![m]\!]^{-1} \sim m$,
(3) if $M \sim m$ then M terminal *if and only if $m \mapsto^*_{\mathbf{do}\ \mathrm{enter}\ \mathbf{as}\ \lambda} m'$* terminal*,*
(4) if $M \sim m$ and $M \mapsto^* M'$ then $m \mapsto^* m'$ such that $M' \sim m'$, and
(5) if $M \sim m$ and $m \mapsto^* m'$ then $M \mapsto^* M'$ such that $M' \sim m'$.

Theorem 6.5 (Operational Correspondence). *For any closed M, $M \mapsto^* M'$* terminal *if and only if $\langle AM[\![M]\!]^{\mathrm{run}}_{\bullet} \rangle\ [\mathrm{run}]\, [\bullet] \mapsto^* m'$* terminal*, and in such a case, $M' \sim m'$.*

6.4 Type system, safety, and erasure

Decompilation does more than relate dynamic semantics; it also relates static semantics of the two as well. If the source program happens to be well-typed, that information is preserved in the machine and can be reflected back. Well-typed programs correspond to well-typed abstract machine configurations—following the typing rules given in fig. 14—with their own type safety property. The key to typing the machine is to understand the two levels of environments corresponding to $\rho\kappa$ versus σ in $c[\rho\kappa]\sigma$. The free variables Γ and results Φ in c refer to registers bound by ρ and κ, while the references out of ρ and κ refer to a surrounding environment Ψ and Ξ bound by σ. From there, we get type safety for the machine that is equivalent to typing in the source.

Assumption 6.6. (1) If $\Psi \vdash \langle f \parallel k \parallel W...\rangle : \Phi$ and $V...; \rho = Load_f(k, W...)$ then $\Psi \vdash \rho : Sig_f(k)$ and $Sig_f(k) \vdash f(k[V...]) : \Phi$.
(2) If $\Psi \vdash \langle s \| W... \| g\rangle : \Phi$ and $V...; \rho = Load_g(s, W...)$ then $\Psi \vdash \rho : Sig_f(s)$ and $Sig_g(s) \vdash g(s[V...]) : \Phi$.

Theorem 6.7 (Type Preservation). *(1) If* m OK *then* $\bullet \vdash AM[\![m]\!]^{-1}$: void : run **comp**.
(2) If $\Gamma \vdash M : B : O$ **comp** *then* $\Gamma \vdash AM[\![M]\!]^{O}_{\Gamma} : B : O$ **comp**.

Lemma 6.8 (Progress & Preservation). *If* m OK *then* $m \mapsto m'$ OK *or* m terminal.

The ty registers in the machine are helpful for maintaining the type preservation link, as they keep track of how generic type variables are instantiated as the program runs. However, they have no impact on the behavior of the machine or the overall result of a program. Note that in fig. 11, the only time the machine reads a ty register is for the purpose of loading another ty register, and the contents of ty registers cannot affect the result of a primitive operation by assumptions 3.1 and 6.3. Therefore, we can erase all types in the program without changing the answer.

Theorem 6.9 (Type Erasure). *Let* erased *be a type constant and let* terminal *states be similar, written* $m \simeq m'$*, if they share the same primitive operation and shape.* $\langle M\rangle\ [\rho\kappa]\,[\sigma]\,[T/\text{ty}\ x...] \mapsto^* m_1$ terminal *if and only if* $\langle M\rangle\ [\rho\kappa]\,[\sigma]\,[\text{erased}/\text{ty}\ x...] \mapsto^* m_2$ terminal *such that* $m_1 \simeq m_2$.

7 FUTURE AND RELATED WORK

Optimizing unboxed data and curried functions. Call-By-Unboxed-Value follows [47]'s tradition of modeling a value's boxed versus unboxed status as a feature in a compiler's intermediate language. This idea was extended to allow for polymorphism over representation of values [19] and the calling convention of functions [15]. Call-By-Unboxed-Value stays closer to more modest roots [17] by keeping representations simple and monomorphic, yet is still able to express many programs that abstract over types with different representations (section 4). Still, there may yet be applications that want to abstract over representations or observations, which we leave to future work.

By decoupling the four-way split between atomic versus complex and value versus computation, Call-By-Push-Value gives a platform for expressing optimizations for curried functions, too. These optimizations are important in practice to avoid wastefully allocating intermediate closures [9, 30, 35]. Usually, the question of how many arguments a function "really" requires (*i.e.,* it's *arity*) is an informal property from complex compile-time analysis [9, 48, 54] and can be easily changed by program optimizations [24]. Call-By-Unboxed takes a type-based approach à la [15, 17] where a function's arity is a property of its type, not just its code. One issue we do not capture here is closure conversion [5, 28]. More recent approaches to typed closure conversion [2, 38] represent them abstractly [8], which has also been modeled in a Call-By-Push-Value framework [50].

Adjoint calculi. Call-By-Unboxed-Value is explicitly inspired by *adjoint calculi* [31, 32, 42, 43, 55, 56]. In essence, these calculi are similar to the monadic framework of computation [41], but they

explicitly divide the program into two parts—positive versus negative, values versus computations, eager versus lazy—in the same way that a monad can be decomposed into an adjoint pair of functors. An advantage of this decomposition is the ability to accurately express the semantics of types, such as "strong sums" [44], especially in the presence of side effects [33], making good on the promise that even effectful programs have the expected isomorphisms between types [34] and can be losslessly compiled down to basic, finite, building blocks [13, 14]. Combining multiple evaluation orders in the same program makes it possible to represent programs that are seemingly polymorphic over evaluation order when the result is the same either way [15, 19] or may be different [18].

Memoization. The interplay between call-by-name and call-by-value is motivated by multiple foundations in denotational semantics and polarized logic. But in practice, non-strict functional languages use *call-by-need* [6, 7] evaluation to *memoize* (*i.e.,* remember) answers and avoid re-computation. As such, we cannot simply "evaluate" a memoized computation; somewhere the answer must be recorded for efficient future retrieval. Call-By-Push-Value has been extended with call-by-need, but at the cost of losing η equalities [37] or an explicitly type-based semantics [13].

Promisingly, we already have a mechanism to talk about different observations—*i.e.,* representations of evaluation contexts—that gives a direct path to insert memoization as another kind of atomic computation different from a simple subroutine (sub **comp**). The evaluation of *memoizing computation* (memo **comp**) is always represented as *two* references memo $x, \overline{x}$: one ($\overline{x}$) to the stack frame needing the answer, and another (x) to the thunk itself to be overwritten. More concretely, we could add memoizing computations as *tagless* [46] thunks to Call-By-Unboxed-Value machine:

$$\frac{\Gamma|_{\Gamma'} \vdash M : B : \mathrm{sub}\ \mathbf{comp}}{\Gamma \vdash \mathrm{start}\ M\,[\Gamma'] : \mathrm{Tape}\ B : \mathrm{ref}\ \mathbf{val}} \qquad \frac{\Gamma \vdash \mathrm{ref}\ x : \mathrm{Tape}\ B : \mathrm{ref}\ \mathbf{val}}{\Gamma \vdash x.\mathrm{play} : B : \mathrm{memo}\ \mathbf{comp}} \qquad \frac{\Gamma|_{\Gamma'} \vdash M : \mathrm{sub}\ \mathbf{comp}}{\Gamma \vdash \mathrm{pause}\ M\,[\Gamma'] : \mathrm{memo}\ \mathbf{comp}}$$

$$\rho^*(V\ldots, \mathrm{start}\ M[\Gamma]) = W\ldots, \mathrm{ref}\ x; x := \mathrm{start}\ M[\rho|_\Gamma] \qquad (\rho^*(V\ldots) = W\ldots; \sigma)$$
$$\langle x.\mathrm{play}\rangle\,[\rho\ \mathrm{sub}\ \overline{x}]\,[\sigma] \mapsto \langle M\rangle\,[\rho'\ \mathrm{memo}\ x, \overline{x}]\,[\sigma] \qquad (\sigma(\rho(\mathrm{ref}\ x)) = \mathrm{start}\ M[\rho'])$$
$$\langle \mathrm{pause}\ M[\Gamma]\rangle\,[\rho\ \mathrm{memo}\ x, \overline{x}]\,[\sigma] \mapsto \langle M\rangle\,[\rho\ \mathrm{sub}\ \overline{x}]\,[\sigma, x := \mathrm{start}(\mathrm{pause}\ M[\Gamma])\,[\rho|_\Gamma]]$$

Tagless thunks are like a cassette tape: they start at the beginning, and when forced, begin to play out until they reach the end when the answer is ready. The tape then stays paused at this end position on all future access. Unlike usual presentations, the program is given control over when to pause the Tape. Call-By-Unboxed-Value could be a good setting to explore mechanisms for memoization—including tagged and tagless styles—and their optimizations, even letting a compiler choose exactly when and how memoization happens depending on the specific application.

Type-safe coercions. Sometimes two different types will have identical representations or calling conventions at runtime, like the unboxed sum examples in section 4. Yet, we intentionally prevent programs of these kinds of seemingly equivalent types from being considered interchangeable; this is part of the reason that compilation to the machine model preserves types. However, there could be many scenarios where it *would* be better to treat two different types as equivalent, such as using an uncurried function $(P_0 \times P_1) \to Q$ in place of a curried one $P_0 \to (P_1 \to Q)$, without any runtime overhead. This is justified because the two different types of call stacks have a one-to-one correspondence with the flattened sequence of atomic arguments in the same order.

In lieu of complex representations [19] and calling conventions [15] to calculate when they are the same at run-time, we could instead employ the more general solution of type-safe coercions [51] to add extra equations between types when their programs are interchangeable. This would make it possible to add other type equalities about unboxed sums—justified when the shapes s and k are compiled into a simple enumeration in a predictable, left-to-right ordering—as in section 4 such as

$(P_0 + P_1) \to Q \approx (P_0 \to Q)\ \&\ (P_1 \to Q)$ and $(P_0 + P_1) \times P_2 \approx (P_0 \times P_2) + (P_1 \times P_2)$, while keeping order-changing inequalities like $P_0 + P_1 \not\approx P_1 + P_0$ separate.

Join points. Sharing code is an important concern in practical implementations, especially when the representation forces the program to do the same work in multiple possible branches [27]. There are multiple approaches to this problem, the most popular being *Static Single Assignment* (SSA) [12] for imperative programs and *Continuation-Passing Style* (CPS) [4] for functional ones, which are known to be related [10, 26]. Another way to share code is with *join points* [36] that keep functional programs in direct style. Extending Call-By-Unboxed-Value with join points would alleviate code duplication problems caused by the mandate to pattern match on complex structures and stacks even if the answer is the same, as we saw in the *and* example in section 4. The code in *and* is small enough to not matter, but in larger examples this doubling is unacceptable. A potential avenue for integrating join points may be a look at the Calculus of Unity [55] which is primarily concerned about naming code, not values; both it and the predecessor to functional join points [16] share a common foundation in the sequent calculus [20] in the style of [11, 53], which could be the key.

Effective dependent types. To be clear, studying dependent types is *not* an objective of this paper. Types are treated as regular first-class, atomic values (represented as erasable phantom ty registers) of type Type τ simply because it is easier if they are not special: the parameter list in an unboxed call stack is just a sequence of values, rather than some interleaving of types and values. The same convenience is used in practice by GHC's Core representation for similar reasons. This simplifying assumption makes it easier formalize quantifiers as $\forall R\ x : A.\ Q$ and $\exists R\ x : A.\ P$ for a generic atomic value type A. Even so, the only interesting choice is to quantify over ty variables, since they are the only ones we are allowed to meaningfully use in the types P or Q (via the *TyVar* rule in fig. 6).

But what if types could refer to other kinds of atomic values, and not just other Type τ parameters? It seems like the natural expression of type abstraction and the quantifiers lends itself readily to a dependently typed calculus. We take pause here and do not jump in eagerly, because the adjoint foundation of Call-By-Unboxed-Value is fundamentally engineered to handle computational effects, and the mixture of effects and dependent types is notoriously fraught with danger [25, 45]. Despite this, there have been some promising starts based on Call-By-Push-Value [45, 52] and sequent calculi [39, 40, 49]. Call-By-Unboxed-Value could be particularly interesting in this space, since it would allow for a richer type system for describing type-safe, low-level representations. What if the programmer wants first-class access to the tags in a tagged union (*i.e.,* unboxed sum type) and control pattern matching? That could be expressed by $\exists\, \mathrm{int}\ x : \mathrm{Nat}\ .\ P$.

8 CONCLUSION

Here, we have introduced the Call-By-Unboxed-Value paradigm, which further decomposes Call-By-Push-Value and focusing regimes based on an operational semantics distinguishing boxed versus unboxed values in real machines. Our goal is to give a more robust foundation for studying the combination of parametric polymorphism with the representation of values and the calling conventions of higher-order functions. It turns out many motivating examples of representation polymorphism can be expressed with a more modest type system, and in fact, representation-irrelevant polymorphic code can be compiled and run without any type information. We hope this enables the study of new applications and implementations of representation irrelevance in other settings. The strength of this approach is to pursue a fine-grained set of tools that can be recombined in new ways. Sometimes the parts are greater than the sum.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under Grant No. 2245516.

Extended syntax:

$$\begin{array}{rlrl} \mathit{StructShape} \ni s & ::= \cdots \mid \square \in \{\, s\ldots \,\} & \mathit{StackShape} \ni k & ::= \cdots \mid \mathrm{more} \in \{\, k\ldots \,\} \\ \mathit{Struct} \ni S & ::= s[\delta] & \mathit{Stack} \ni K & ::= k[\delta] \\ \mathit{Values} \ni \delta & ::= \bullet \mid \delta, V \mid \delta, S & \mathit{Call} \ni L & ::= \cdots \mid M \end{array}$$

$$\Gamma ::= \cdots \mid \Gamma, x : P : \mathbf{cplx\ val} \qquad \Delta ::= \cdots \mid P : \mathbf{cplx\ val}, \Delta \qquad \Phi ::= \cdots \mid Q : \mathbf{cplx\ comp}$$

$$\frac{\forall(\Gamma \mid \Delta \vdash s : P\ ;)}{\Gamma, x : P\ \mathbf{cplx\ val}, \Gamma' \vdash x \in \{\, s \overset{s \in P}{\ldots} \,\} : P} \qquad \frac{\forall(\Gamma \mid \Delta \vdash s : P\ ;)}{\Gamma \mid P : \mathbf{cplx\ val} \vdash \square \in \{\, s \overset{s \in P}{\ldots} \,\} : P\ ;}$$

$$\frac{\Gamma \mid \Delta \vdash s : P\ ; \quad \Gamma \vdash \delta : \Delta}{\Gamma \vdash s[\delta] : P}\ \mathit{Struct} \qquad \frac{\Gamma \mid \Delta\ ; k : Q \vdash \Phi \quad \Gamma \vdash \delta : \Delta}{\Gamma \mid k[\delta] : Q \vdash \Phi}\ \mathit{Stack}$$

$$\frac{\Gamma \vdash S : P \quad \Gamma \vdash \delta : \Delta}{\Gamma \vdash (S, \delta) : (P : \mathbf{cplx\ val}, \Delta)}$$

$$\frac{\forall(\Gamma \mid \Delta\ ; k : Q \vdash \Phi)}{\Gamma \mid \bullet\ ; \mathrm{more} \in \{\, k \overset{k \in Q}{\ldots} \,\} : Q \vdash Q : \mathbf{cplx\ comp}} \qquad \frac{\Gamma \vdash M : Q : \mathbf{cplx\ comp}}{\Gamma \vdash M : Q}$$

Fig. 15. Complex Call-By-Unboxed-Value: the extension with (co)pattern disjunction.

A COMPLEX VARIABLES IN CALL-BY-UNBOXED-VALUE

Sometimes, being forced to elaborate all pattern-matching options can be rather burdensome when the result is the same in multiple cases. Not only does it waste more bits or ink, it can cause serious code duplication problems. In lieu of a more serious solution, like join points [36], we can easily add some syntactic sugar for letting us assign a name to a whole complex value, corresponding to Zeilberger's *complex variables* [56]. However, [56]'s notion of complex variables are only meaningful in a typed setting: the missing patterns are elaborated by checking the type of the variable and expanding the options. Instead, we still want to be able to compile and run untyped code, even when using this shorthand to combine redundant cases.

Our solution is to extend Call-By-Unboxed-Value with the ability to summarize multiple (co)-patterns within the same alternative branch, as shown in fig. 15. Intuitively, the idea is that we might combine multiple patterns disjunctively, by saying what to do if either "this *or* that" matches. This disjunction can be embedded inside of a larger pattern, in which case we can assign a name to the choice, written as $x \in \{\, s_i \overset{i \in I}{\ldots} \,\}$, where the set $\{\, s_i \overset{i \in I}{\ldots} \,\}$ disambiguates all the possible different shapes that the complex x might take.

Disjunction shouldn't just be limited to pattern variables: it's useful in the result of a call, too. In particular, we might want to only partially specify a complex curried function, writing only some of the λ-abstractions and projection alternatives and then leaving the right-hand side as another complex computation. We can end the partial abstraction early, writing $\mathrm{more} \in \{\, k_i \overset{i \in I}{\ldots} \,\}$, where the set $\{\, k_i \overset{i \in I}{\ldots} \,\}$ disambiguates all the possible ways that the complex call could continue.

The reason to include the disambiguating set for complex variables $x \in \{\, s\ldots \,\}$ and complex continuations $\mathrm{more} \in \{\, k\ldots \,\}$ is to give just enough information that programs can be desugared into the simpler Call-By-Unboxed-Value syntax in fig. 3. This elaboration is shown in fig. 16. Notice that no type information is needed for the macro expansion, so untyped programs can still be compiled and run. This serves as an untyped alternative to the explicitly-typed complex variables of [56]. Moreover, we did not need to complicate the language of representations or observations to

$$\begin{aligned}
\mathit{PatternCxt} \ni p^1 &::= \Box \mid p^1, p \mid p, p^1 \mid b, p^1 \mid R\ x : T, p^1 \\
\mathit{CoPatternCxt} \ni q^1 &::= \Box \mid p^1 \cdot q \mid p \cdot q^1 \mid b \cdot q^1 \mid R\ x : A \cdot q^1
\end{aligned}$$

$$\begin{aligned}
\{\ldots; p^1[x \in \{\, s_i \overset{i\in I}{\ldots} \,\}] \to M\} &\to \{\ldots; p^1[s_i[x_i \ldots]] \to M[s_i[x_i \ldots]/x] \overset{i\in I}{\ldots}\} \\
\{\ldots; q^1[x \in \{\, s_i \overset{i\in I}{\ldots} \,\}] \to M\} &\to \{\ldots; q^1[s_i[x_i \ldots]] \to M[s_i[x_i \ldots]/x] \overset{i\in I}{\ldots}\} \\
\{\ldots; q^1[\mathrm{more} \in \{\, k_i \overset{i\in I}{\ldots} \,\}] \to M\} &\to \{\ldots; q^1[k_i[x_i \ldots]] \to \langle M \parallel k_i[x_i \ldots]\rangle \overset{i\in I}{\ldots}\} \\
\langle S\ \mathbf{as}\ \{\, p \to M_p \overset{p\in P}{\ldots} \,\} \parallel K\rangle &\to S\ \mathbf{as}\ \{\, p \to \langle M_p \parallel K\rangle \overset{p\in P}{\ldots} \,\} \\
\langle \mathbf{unbox}\ V\ \mathbf{as}\ \{\, p \to M_p \overset{p\in P}{\ldots} \,\} \parallel K\rangle &\to \mathbf{unbox}\ V\ \mathbf{as}\ \{\, p \to \langle M_p \parallel K\rangle \overset{p\in P}{\ldots} \,\} \\
\langle \mathbf{do}\ M\ \mathbf{as}\ \{\, p \to M_p \overset{p\in P}{\ldots} \,\} \parallel K\rangle &\to \mathbf{do}\ M\ \mathbf{as}\ \{\, p \to \langle M_p \parallel K\rangle \overset{p\in P}{\ldots} \,\}
\end{aligned}$$

$$\begin{aligned}
s[\delta] \in \{\, s_i \overset{i\in I}{\ldots} \,\} &\to s[\delta] && (s \in \{\, s_i \overset{i\in I}{\ldots} \,\}) \\
\langle\langle L \parallel q^1[\delta, \mathrm{more} \in \{\, k_i \overset{i\in I}{\ldots} \,\}]\rangle \parallel k[\delta']\rangle &\to \langle L \parallel q^1[\delta, k[\delta']]\rangle && (k \in \{\, k_i \overset{i\in I}{\ldots} \,\})
\end{aligned}$$

Fig. 16. Untyped macro expansion of complex (co)pattern disjunction

do so, either. In that way, the (co)pattern shape sets serve as a more modest alternative to complex, multi-faceted representations [19] and calling conventions [15].

As an example, the shared code in the boolean *and* function

$$\begin{aligned}
&\mathit{and} && :: \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Bool} \\
&\mathit{and}\ \mathrm{True}\ x && = x \\
&\mathit{and}\ \mathrm{False}\ x && = \mathrm{False}
\end{aligned}$$

can be kept in tact using pattern disjunction like so:

$$\begin{aligned}
&\mathit{and} : \mathrm{Bool} \to \mathrm{Bool} \to \mathrm{Eval}(\mathrm{Ret}\ \mathrm{Bool}) \\
&\mathit{and} = \{1, () \cdot x \in \{\, 1, (); \mathtt{0}, () \,\} \cdot \mathrm{eval} \to \mathbf{ret}\ x \\
&\qquad\ \ \mathtt{0}, () \cdot x \in \{\, 1, (); \mathtt{0}, () \,\} \cdot \mathrm{eval} \to \mathbf{ret}\ \mathtt{0}, ()\}
\end{aligned}$$

Desugaring this definition using fig. 16 gives exactly the fully-elaborated, four-way branching version from section 4.

B POLYMORPHIC CALL-BY-PUSH-VALUE λ-CALCULUS

The full definition of the polymorphic Call-By-Push-Value λ-calculus is given in figs. 17 to 20.

Notice that Call-By-Push-Value's sequencing axioms from fig. 20 don't seem to appear in Call-By-Unboxed-Value simple $\beta\eta$ equational theory in section 3.4. These are equivalent to conversions that commute a proc computation to pull out of any block statement—**do**, **unbox**, or a plain **as**—to pop the stack first before running the computation, like so:

$$\begin{aligned}
&(cc\ \mathrm{Proc}) && \mathbf{do}\ M\ \mathbf{as}\ \{\, p \to \mathrm{proc}\ \{\, q \to M'_{qp} \overset{q\in Q}{\ldots} \,\} \overset{p\in P}{\ldots} \,\} = \mathrm{proc}\ \{\, q \to \mathbf{do}\ M\ \mathbf{as}\ \{\, p \to M'_{qp} \overset{p\in P}{\ldots} \,\} \overset{q\in Q}{\ldots} \,\} \\
&(cc\ \mathrm{Proc}) && \mathbf{unbox}\ V\ \mathbf{as}\ \{\, p \to \mathrm{proc}\ \{\, q \to M_{qp} \overset{q\in Q}{\ldots} \,\} \overset{p\in P}{\ldots} \,\} = \mathrm{proc}\ \{\, q \to \mathbf{unbox}\ V\ \mathbf{as}\ \{\, p \to M_{qp} \overset{p\in P}{\ldots} \,\} \overset{q\in Q}{\ldots} \,\} \\
&(cc\ \mathrm{Proc}) && S\ \mathbf{as}\ \{\, p \to \mathrm{proc}\ \{\, q \to M_{qp} \overset{q\in Q}{\ldots} \,\} \overset{p\in P}{\ldots} \,\} = \mathrm{proc}\ \{\, q \to S\ \mathbf{as}\ \{\, p \to M_{qp} \overset{p\in P}{\ldots} \,\} \overset{q\in Q}{\ldots} \,\}
\end{aligned}$$

These equations are derivable from the $\beta\eta$ axioms seen already in figs. 5 and 9.

$$
\begin{array}{rcl}
\mathit{Kind} \ni \tau & ::= & \mathbf{val} \mid \mathbf{comp} \\
\mathit{Type} \ni T & ::= & A \mid \underline{B} \\
\mathit{ValueType} \ni A & ::= & X \mid 1 \mid A_0 \times A_1 \mid 0 \mid A_0 + A_1 \mid \exists X : \tau . A \mid \mathrm{U}\, \underline{B} \\
\mathit{CompType} \ni \underline{B} & ::= & X \mid A \to \underline{B} \mid \top \mid \underline{B}_0 \,\&\, \underline{B}_1 \mid \forall X : \tau . \underline{B} \mid \mathrm{F}\, A
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Value} \ni V & ::= & x \mid () \mid (V_0, V_1) \mid (\mathtt{0}, V) \mid (\mathtt{1}, V) \mid \mathrm{thunk}\, M \\
\mathit{Comp} \ni M & ::= & \mathbf{do}\, x : A \leftarrow M\, \mathbf{in}\, M' \mid \mathbf{return}\, V \mid V.\,\mathrm{force} \\
& & \mid \mathbf{match}\, V\, \mathbf{as}() \to M \mid \mathbf{match}\, V\, \mathbf{as}(x_0 : A_0, x_1 : A_1) \to M \\
& & \mid \mathbf{match}\, V\, \mathbf{as}\, \{\, (b, x_b : A_b) \to M_b{}^{b \in \{0,1\}}_{\cdots} \,\} \mid \mathbf{match}\, V\, \mathbf{as}(X : \tau, x : A) \to M \\
& & \mid \lambda x{:}A.M \mid M\, V \mid \lambda\, \{\, \} \mid \lambda\, \{\, b.M^{b \in \{0,1\}}_{\cdots} \,\} \mid M\, \mathtt{0} \mid M\, \mathtt{1} \mid \Lambda X{:}\tau.M \mid M\, T
\end{array}
$$

Fig. 17. Polymorphic Call-By-Push-Value syntax.

$$
\mathit{EvalCxt} \ni E ::= \Box \mid \mathbf{do}\, x : A \leftarrow E\, \mathbf{in}\, M \mid E\, V \mid E\, \mathtt{0} \mid E\, \mathtt{1} \mid E\, T
$$

$$
\begin{array}{rr}
(\beta\,\mathrm{F}) & \mathbf{do}\, x : A \leftarrow \mathbf{return}\, V\, \mathbf{in}\, M \mapsto M[V/x] \\
(\beta 1) & \mathbf{match}\, ()\, \mathbf{as}\, \{\, () \to M \,\} \mapsto M \\
(\beta\times) & \mathbf{match}\, (V_0, V_1)\, \mathbf{as}\, \{\, (x_0 : A_0, x_1 : A_1) \to M \,\} \mapsto M[V_0/x_0, V_1/x_1] \\
(\beta{+}_0) & \mathbf{match}\, (\mathtt{0}, V)\, \mathbf{as}\, \{\, (b, x_b : A_b) \to M_b{}^{b \in \{0,1\}}_{\cdots} \,\} \mapsto M_0[V/x_0] \\
(\beta{+}_1) & \mathbf{match}\, (\mathtt{1}, V)\, \mathbf{as}\, \{\, (b, x_b : A_b) \to M_b{}^{b \in \{0,1\}}_{\cdots} \,\} \mapsto M_1[V/x_1] \\
(\beta\exists) & \mathbf{match}\, (T, V)\, \mathbf{as}\, \{\, (X : \tau, x : A) \to M \,\} \mapsto M[T/X, V/x] \\
(\beta\,\mathrm{U}) & (\mathrm{thunk}\, M).\,\mathrm{force} \mapsto M \\
(\beta\to) & (\lambda x{:}A.M)\, V \mapsto M[V/x] \\
(\beta\&_0) & (\lambda\, \{\, b.M_b{}^{b \in \{0,1\}}_{\cdots} \,\})\, \mathtt{0} \mapsto M_0 \\
(\beta\&_1) & (\lambda\, \{\, b.M_b{}^{b \in \{0,1\}}_{\cdots} \,\})\, \mathtt{1} \mapsto M_1 \\
(\beta\forall) & (\Lambda X{:}\tau.M)\, T \mapsto M[T/X]
\end{array}
$$

Fig. 18. Polymorphic Call-By-Push-Value operational semantics.

If we want to look at things the other way, we can likewise push the stack frames downward into block statements, toward the sub-procedures that want them, like so:

$$
\begin{array}{rr}
(cc\ \mathrm{enter}) & \langle \mathbf{do}\, M\, \mathbf{as}\, \{\, p \to M_p{}^{p \in P}_{\cdots} \,\}.\,\mathrm{enter}\, \|K\rangle = \mathbf{do}\, M\, \mathbf{as}\, \{\, p \to \langle M_p.\,\mathrm{enter}\, \|K\rangle{}^{p \in P}_{\cdots} \,\} \\
(cc\ \mathrm{enter}) & \langle \mathbf{unbox}\, V\, \mathbf{as}\, \{\, p \to M_p{}^{p \in P}_{\cdots} \,\}.\,\mathrm{enter}\, \|K\rangle = \mathbf{unbox}\, V\, \mathbf{as}\, \{\, p \to \langle M_p.\,\mathrm{enter}\, \|K\rangle{}^{p \in P}_{\cdots} \,\} \\
(cc\ \mathrm{enter}) & \langle S\, \mathbf{as}\, \{\, p \to M_p{}^{p \in P}_{\cdots} \,\}.\,\mathrm{enter}\, \|K\rangle = S\, \mathbf{as}\, \{\, p \to \langle M_p.\,\mathrm{enter}\, \|K\rangle{}^{p \in P}_{\cdots} \,\}
\end{array}
$$

Notice that the *cc* enter and *cc* Proc rules are inter-derivable via the $\beta\eta$ Proc rules.

In their most general form, we can summarize *all* of these small-step commutations by a single commutation between *tail contexts* C^{tl} with the introduction and elimination forms of the Proc Q

Kinds of types $\boxed{\Gamma \vdash T : \tau}$

$$\frac{}{\Gamma, X : \tau, \Gamma' \vdash X : \tau}\ \mathit{TyVar} \qquad \frac{}{\Gamma \vdash 1 : \mathbf{val}}\ 1T \qquad \frac{}{\Gamma \vdash 0 : \mathbf{val}}\ 0T \qquad \frac{}{\Gamma \vdash \top : \mathbf{comp}}\ \top T$$

$$\frac{\Gamma \vdash A_0 : \mathbf{val} \quad \Gamma \vdash A_1 : \mathbf{val}}{\Gamma \vdash A_0 \times A_1 : \mathbf{val}}\ \times T \qquad \frac{\Gamma \vdash A_0 : \mathbf{val} \quad \Gamma \vdash A_1 : \mathbf{val}}{\Gamma \vdash A_0 + A_1 : \mathbf{val}}\ + T$$

$$\frac{\Gamma, X : \tau \vdash A : \mathbf{val}}{\Gamma \vdash \exists X : \tau . A : \mathbf{val}}\ \exists T \qquad \frac{\Gamma \vdash \underline{B} : \mathbf{comp}}{\Gamma \vdash \mathrm{U}\,\underline{B} : \mathbf{val}}\ \mathrm{U}T$$

$$\frac{\Gamma \vdash A : \mathbf{val} \quad \Gamma \vdash \underline{B} : \mathbf{comp}}{\Gamma \vdash A \to \underline{B} : \mathbf{comp}}\ \to T \qquad \frac{\Gamma \vdash \underline{B}_0 : \mathbf{comp} \quad \Gamma \vdash \underline{B}_1 : \mathbf{comp}}{\Gamma \vdash \underline{B}_0 \,\&\, \underline{B}_1 : \mathbf{comp}}\ \& T$$

$$\frac{\Gamma, X : \tau \vdash \underline{B} : \mathbf{comp}}{\Gamma \vdash \forall X : \tau . \underline{B} : \mathbf{comp}}\ \forall T \qquad \frac{\Gamma \vdash A : \mathbf{val}}{\Gamma \vdash \mathrm{F}\,A : \mathbf{comp}}\ \exists T$$

Types of values $\boxed{\Gamma \vdash V : A}$

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ \mathit{Var} \qquad \frac{}{\Gamma \vdash () : 1}\ 1I \qquad \text{No } 0I \text{ rules}$$

$$\frac{\Gamma \vdash V_0 : A_0 \quad \Gamma \vdash V_1 : A_1}{\Gamma \vdash (V_0, V_1) : A_0 \times A_1}\ \times I \qquad \frac{\Gamma \vdash V : A_0}{\Gamma \vdash (0, V) : A_0 + A_1}\ + I_0 \qquad \frac{\Gamma \vdash V : A_1}{\Gamma \vdash (1, V) : A_0 + A_1}\ + I_1$$

$$\frac{\Gamma \vdash T : \tau \quad \Gamma \vdash V : A[T/X]}{\Gamma \vdash (T, V) : \exists X : \tau . A}\ \exists I \qquad \frac{\Gamma \vdash M : \underline{B}}{\Gamma \vdash \mathrm{thunk}\, M : \mathrm{U}\,\underline{B}}\ \mathrm{U}I$$

Types of computations $\boxed{\Gamma \vdash M : \underline{B}}$

$$\frac{\Gamma \vdash M : \mathrm{F}\,A \quad \Gamma, x : A \vdash M' : \underline{B}}{\Gamma \vdash \mathbf{do}\, x{:}A \leftarrow M\, \mathbf{in}\, M' : \underline{B}}\ \mathrm{F}E \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{return}\, V : \mathrm{F}\,A}\ \mathrm{F}I$$

$$\frac{\Gamma \vdash V : 1 \quad \Gamma \vdash M : \underline{B}}{\Gamma \vdash \mathbf{match}\, V\, \mathbf{as}\, () \to M : \underline{B}}\ 1E \qquad \frac{\Gamma \vdash V : A_0 \times A_1 \quad \Gamma, x_0 : A_0, x_1 : A_1 \vdash M : \underline{B}}{\Gamma \vdash \mathbf{match}\, V\, \mathbf{as}\, (x_0 : A_0, x_1 : A_1) \to M : \underline{B}}\ \times E$$

$$\frac{\Gamma \vdash V : 0}{\Gamma \vdash \mathbf{match}\, V\, \mathbf{as}\, \{\ \} : \underline{B}}\ 0E \qquad \frac{\Gamma \vdash V : A_0 + A_1 \quad \Gamma, x_0 : A_0 \vdash M_0 : \underline{B} \quad \Gamma, x_1 : A_1 \vdash M_1 : \underline{B}}{\Gamma \vdash \mathbf{match}\, V\, \mathbf{as}\, \{\ (0, x_0 : A_0) \to M_0; (1, x_1 : A_1) \to M_1\ \} : \underline{B}}\ \& E$$

$$\frac{\Gamma \vdash V : \exists X : \tau . A \quad \Gamma, X : \tau, x : A \vdash M : \underline{B}}{\Gamma \vdash \mathbf{match}\, V\, \mathbf{as}\, (X : \tau, x : A) \to M : \underline{B}}\ \times E \qquad \frac{\Gamma \vdash V : \mathrm{U}\,\underline{B}}{\Gamma \vdash V . \mathrm{force} : \underline{B}}\ \mathrm{U}E$$

$$\frac{\Gamma, x : A \vdash M : \underline{B}}{\Gamma \vdash \lambda x{:}A . M : A \to \underline{B}}\ \to I \qquad \frac{\Gamma \vdash M : A \to \underline{B} \quad \Gamma \vdash V : A}{\Gamma \vdash M\, V : \underline{B}}\ \to E$$

$$\frac{\Gamma \vdash M_0 : \underline{B}_0 \quad \Gamma \vdash M_1 : \underline{B}_1}{\Gamma \vdash \lambda\, \{\, 0 . M_0; 1 . M_1\, \} : \underline{B}_0 \,\&\, \underline{B}_1}\ \& I \qquad \frac{\Gamma \vdash M : \underline{B}_0 \,\&\, \underline{B}_1}{\Gamma \vdash M\, 0 : \underline{B}_0}\ \& E_0 \qquad \frac{\Gamma \vdash M : \underline{B}_0 \,\&\, \underline{B}_1}{\Gamma \vdash M\, 1 : \underline{B}_1}\ \& E_1$$

$$\frac{}{\Gamma \vdash \lambda\, \{\ \} : \top}\ \top I \qquad \text{No } \top E \text{ rules.} \qquad \frac{\Gamma, X : \tau \vdash M : \underline{B}}{\Gamma \vdash \Lambda X{:}\tau . M : \forall X{:}\tau . \underline{B}}\ \forall I \qquad \frac{\Gamma \vdash M : \forall X{:}\tau . \underline{B} \quad \Gamma \vdash T : \tau}{\Gamma \vdash M\, T : \underline{B}[T/X]}\ \forall E$$

Fig. 19. Polymorphic Call-By-Push-Value type system.

Rules for congruence (equality can be applied in any context) plus:

$$\frac{\Gamma \vdash M : \underline{B}}{\Gamma \vdash M = M : \underline{B}}\ \mathit{Refl} \qquad \frac{\Gamma \vdash M = M' : \underline{B}}{\Gamma \vdash M' = M : \underline{B}}\ \mathit{Symm} \qquad \frac{\Gamma \vdash M = M' : \underline{B} \quad \Gamma \vdash M' = M'' : \underline{B}}{\Gamma \vdash M = M'' : \underline{B}}\ \mathit{Trans}$$

$$\frac{\Gamma \vdash M = M' : \underline{B} \quad M' \mapsto M''}{\Gamma \vdash M = M'' : \underline{B}}\ \mathit{Step}$$

Extensional η axioms:

$$\begin{array}{lrll}
(\eta{\to}) & \lambda x{:}A.(M\ x) = M & : A \to \underline{B} & (x \notin FV(M)) \\
(\eta 0) & \lambda\,\{\ \} = M & : \top & \\
(\eta \&) & \lambda\,\{\ 0.(M\ 0); 1.(M\ 1)\ \} = M & : \underline{B}_0\ \&\ \underline{B}_1 & \\
(\eta \forall) & \Lambda X{:}\tau.(M\ X) = M & : \forall X{:}\tau.\underline{B} & (X \notin FV(M)) \\
(\eta\, \mathrm{U}) & \mathrm{thunk}(M.\,\mathrm{force}) = M & : \mathrm{U}\,\underline{B} & \\
(\eta\, \mathrm{F}) & \mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\ \mathbf{return}\ x = M & : \mathrm{F}\,A & \\
(\eta 1) & \mathbf{match}\ V\ \mathbf{as}\ \{\ () \to M[()/x]\ \} = M[V/x] & (V : 1) & \\
(\eta{\times}) & \mathbf{match}\ V\ \mathbf{as}(x_0{:}A_0, x_1{:}A_1) \to M[(x_0, x_1)/x] = M[V/x] & (V : A_0 \times A_1, & x_0, x_1 \notin FV(M)) \\
(\eta 0) & \mathbf{match}\ V\ \mathbf{as}\ \{\ \} = M[V/x] & (V : 0) & \\
(\eta{+}) & \mathbf{match}\ V\ \mathbf{as}\{(b, x_b{:}A_b) \to M_b[(b, x_b)/x]^{b \in \{0,1\}}_{\cdots}\} = M[V/x] & (V : A_0 + B_0, & x_b \notin FV(M_b)) \\
(\eta{\times}) & \mathbf{match}\ V\ \mathbf{as}(X{:}\tau, x{:}A) \to M[(X, x)/x] = M[V/x] & (V : \exists X{:}\tau.A, & X, x \notin FV(M))
\end{array}$$

Sequencing axioms:

$$\begin{array}{lll}
(cc{\to}) & \begin{array}{l}\mathbf{do}\ x : A \leftarrow M \\ \mathbf{in}\ (\lambda y : A'.M')\end{array} = \lambda y : A'.(\mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\ M') & (y \notin FV(M)) \\
(cc\&) & \begin{array}{l}\mathbf{do}\ x : A \leftarrow M \\ \mathbf{in}\ (\lambda\,\{\ b.\ M_b{}^{b \in \{0,1\}}_{\cdots}\ \})\end{array} = \lambda\,\{\ b.(\mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\ M_b)^{b \in \{0,1\}}_{\cdots}\ \} & \\
(cc\forall) & \begin{array}{l}\mathbf{do}\ x : A \leftarrow M \\ \mathbf{in}\ (\Lambda X : \tau.M')\end{array} = \Lambda X : \tau.(\mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\ M') & (X \notin FV(M)) \\
(cc\,\mathrm{F}) & \begin{array}{l}\mathbf{do}\ x : A \leftarrow M\ \mathbf{in} \\ \mathbf{do}\ y : A' \leftarrow M' \\ \mathbf{in}\ M''\end{array} = \begin{array}{l}\mathbf{do}\ x : A \leftarrow (\ \mathbf{do}\ y : A \leftarrow M' \\ \qquad\qquad \mathbf{in}\ M) \\ \mathbf{in}\ M''\end{array} & (x \notin FV(M'), y \notin FV(M))
\end{array}$$

Fig. 20. Polymorphic Call-By-Push-Value equational theory.

type, expressed by just these two axioms:

$$\begin{array}{rl}
\mathit{TailCxt} \ni C^{tl} & ::= \Box \mid S\ \mathbf{as}\ C^{mr} \mid \mathbf{unbox}\ V\ \mathbf{as}\ C^{mr} \mid \mathbf{do}\ M\ \mathbf{as}\ C^{mr} \\
\mathit{MatchRespCxt} \ni C^{mr} & ::= \{\ p \to C^{tl}_{p}{}^{p \in P}_{\cdots}\ \} \\
(cc\,\mathrm{proc}) & C^{tl}[\mathrm{proc}\,\{\ q \to M_{iq}{}^{q \in Q}_{\cdots}\ \}^{i \in I}_{\cdots}] = \mathrm{proc}\,\{\ q \to C^{tl}[M_{iq}{}^{i \in I}_{\cdots}]^{q \in Q}_{\cdots}\ \} \\
(cc\,\mathrm{enter}) & \langle C^{tl}[M_i{}^{i \in I}_{\cdots}].\,\mathrm{enter}\ \|K\rangle = C^{tl}[\langle M_i.\,\mathrm{enter}\ \|K\rangle^{i \in I}_{\cdots}]
\end{array}$$

which can be derived from the small-step commutations by induction on C^{tl}.

REFERENCES

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2429069.2429075

[2] Amal Ahmed and Matthias Blume. 2008. Typed closure conversion preserves observational equivalence. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 157–168.

[3] Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2, 3 (1992), 297–347. https://doi.org/10.1093/logcom/2.3.297

[4] Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press.

[5] Andrew W. Appel and Trevor Jim. 1989. Continuation-Passing, Closure-Passing Style. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. 293–302.

[6] Zena M. Ariola and Matthias Felleisen. 1997. The Call-By-Need Lambda Calculus. *Journal of Functional Programming* 7, 3 (May 1997), 265–301. https://doi.org/10.1017/S0956796897002724

[7] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-By-Need Lambda Calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. ACM, New York, NY, USA, 233–246.

[8] William J. Bowman and Amal Ahmed. 2018. Typed closure conversion for the calculus of constructions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 797–811.

[9] Joachim Breitner. 2014. Call Arity. In *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*. 34–50.

[10] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. 2003. A Functional Perspective on SSA Optimisation Algorithms. *Electr. Notes Theor. Comput. Sci.* 82, 2 (2003), 347–361. https://doi.org/10.1016/S1571-0661(05)82596-4

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

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (1991), 451–490.

[13] Paul Downen and Zena M. Ariola. 2018. Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*. 21:1–21:23.

[14] Paul Downen and Zena M. Ariola. 2020. Compiling With Classical Connectives. *Log. Methods Comput. Sci.* 16, 3 (2020). https://lmcs.episciences.org/6740

[15] Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. 2020. Kinds are calling conventions. *Proc. ACM Program. Lang.* 4, ICFP (2020), 104:1–104:29. https://doi.org/10.1145/3408986

[16] Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. 2016. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 74–88. https://doi.org/10.1145/2951913.2951931

[17] Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Making a Faster Curry with Extensional Types. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 58–70. https://doi.org/10.1145/3331545.3342594

[18] Joshua Dunfield. 2015. Elaborating evaluation-order polymorphism. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 256–268.

[19] Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 525–539. https://doi.org/10.1145/3062341.3062357

[20] Gerhard Gentzen. 1935. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift* 39, 1 (1935), 176–210. https://doi.org/10.1007/BF01201353

[21] Andy Gill and Graham Hutton. 2009. The worker/wrapper transformation. *Journal of Functional Programming* 19, 2 (2009), 227–251. https://doi.org/10.1017/S0956796809007175

[22] Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph. D. Dissertation. Université Paris 7.

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

[24] John Hannan and Patrick Hicks. 1998. Higher-Order Arity Raising. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*. 27–38.

[25] Hugo Herbelin. 2005. On the Degeneracy of Sigma-Types in Presence of Computational Classical Logic. In *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3461)*, Pawel Urzyczyn (Ed.). Springer, 209–220.

[26] Richard Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*. 13–23. https://doi.org/10.1145/202529.202532

[27] Andrew Kennedy. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*. 177–190. https://doi.org/10.1145/1291151.1291179

[28] Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320.

[29] Olivier Laurent. 2002. *Étude de la polarisation en logique*. Ph. D. Dissertation. Université de la Méditerranée - Aix-Marseille II.

[30] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA.

[31] Paul Blain Levy. 2001. *Call-By-Push-Value*. Ph. D. Dissertation. Queen Mary and Westfield College, University of London.

[32] Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation* 19, 4 (01 Dec. 2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

[33] Paul Blain Levy. 2006. *Jumbo λ-Calculus*. Springer Berlin Heidelberg, Berlin, Heidelberg, 444–455. https://doi.org/10.1007/11787006_38

[34] Paul Blain Levy. 2017. Contextual isomorphisms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 400–414. https://doi.org/10.1145/3009837.3009898

[35] Simon Marlow and Simon L. Peyton Jones. 2004. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. ACM, 4–15.

[36] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling Without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. 482–494.

[37] Dylan McDermott and Alan Mycroft. 2019. Extended Call-by-Push-Value: Reasoning About Effectful Programs and Evaluation Order. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 235–262.

[38] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. 271–283.

[39] Étienne Miquey. 2017. *Classical realizability and side-effects. (Réalisabilité classique et effets de bords)*. Ph. D. Dissertation. University of the Republic, Montevideo, Uruguay.

[40] Étienne Miquey. 2019. A Classical Sequent Calculus with Dependent Types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 2 (March 2019), 1–48. https://doi.org/10.1145/3230625

[41] Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (Pacific Grove, California, USA). IEEE Press, Piscataway, NJ, USA, 14–23. http://dl.acm.org/citation.cfm?id=77350.77353

[42] Guillaume Munch-Maccagnoni. 2009. Focalisation and Classical Realisability. In *Computer Science Logic: 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL* (Coimbra, Portugal) *(CSL 2009)*, Erich Grädel and Reinhard Kahle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 409–423.

[43] Guillaume Munch-Maccagnoni. 2013. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. Ph. D. Dissertation. Université Paris Diderot.

[44] Guillaume Munch-Maccagnoni and Gabriel Scherer. 2015. Polarised Intermediate Representation of Lambda Calculus with Sums. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science* (Kyoto, Japan) *(LICS 2015)*. 127–140.

[45] Pierre-Marie Pédrot and Nicolas Tabareau. 2019. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.* 4, POPL, Article 58 (dec 2019), 28 pages. https://doi.org/10.1145/3371126

[46] Simon L. Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming* 2, 2 (1992), 127–202.

[47] Simon L. Peyton Jones and John Launchbury. 1991. Unboxed Values As First Class Citizens in a Non-Strict Functional Language. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*.

Springer-Verlag, London, UK, UK, 636–666.
[48] Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Modular, Higher-order Cardinality Analysis in Theory and Practice. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. 335–347.
[49] Arnaud Spiwack. 2014. A Dissection of L. https://assert-false.science/arnaud/papers/A%20dissection%20of%20L.pdf
[50] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. 2023. Closure Conversion in Little Pieces. In *International Symposium on Principles and Practice of Declarative Programming, PPDP 2023, Lisboa, Portugal, October 22-23, 2023*, Santiago Escobar and Vasco T. Vasconcelos (Eds.). ACM, 10:1–10:13. https://doi.org/10.1145/3610612.3610622
[51] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*. 53–66.
[52] Matthijs Vákár. 2017. *In Search of Effectful Dependent Types*. Ph. D. Dissertation. Magdalen College, University of Oxford.
[53] Philip Wadler. 2003. Call-By-Value is Dual to Call-By-Name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden). ACM, New York, NY, USA, 189–201. https://doi.org/10.1145/944705.944723
[54] Dana N. Xu and Simon L. Peyton Jones. 2005. Arity Analysis. (2005). Working notes.
[55] Noam Zeilberger. 2008. On the Unity of Duality. *Annals of Pure and Applied Logic* 153, 1 (2008), 660–96.
[56] Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching*. Ph. D. Dissertation. Carnegie Mellon University.

Received 28 February 2024; revised 10 June 2024