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. *J. ACM* 37, 4, Article 111 (September 2024), 56 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, 46] 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 1557-735X/2024/9-ART111
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 [30] 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. In the beginning, Peyton Jones and Launchbury [46] observed that unboxed values *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. Functions expecting input in a floating-point register cannot be given a thunk address. So even a lazy language needs to strictly pass unboxed arguments by value.

Surprisingly, the indirection cost of a lazy floating-point number is reflected in denotational semantics: the domain of efficient unboxed numbers must be *unlifted*. So in the interest of finding a logical foundation for unboxed representations, we should search for semantics that lets us avoid lifting as much as possible. One such option is Call-By-Push-Value [30]. Designed to subsume both call-by-name and call-by-value, Call-By-Push-Value avoids implicit lifts, since they are easy to add but hard to remove. This is achieved by a strict distinction between values that *already are* versus computations that *will do*, showing up as two different kinds of types:

$$\begin{aligned} \mathit{ValueType} \ni A &::= 1 \mid A_0 \times A_1 \mid 0 \mid A_0 + A_1 \mid \mathrm{U}\,\underline{B} \\ \mathit{ComputationType} \ni \underline{B} &::= A \to \underline{B} \mid \top \mid \underline{B}_0 \,\&\, \underline{B}_1 \mid \mathrm{F}\,A \end{aligned}$$

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. It is seen again in the Calculus of Unity [54]—an interpretation of proof-theoretic *focusing* [3, 28] as nested pattern matching—for completely different reasons. A key step of focusing is to recognize *positive* versus *negative* types, divided like so:

$$\begin{aligned} \mathit{PositiveType} \ni P^+ &::= 1 \mid P_0^+ \otimes P_1^+ \mid 0 \mid P_0^+ \oplus P_1^+ \mid {\downarrow}Q^- \\ \mathit{NegativeType} \ni Q^- &::= P^+ \to Q^- \mid \top \mid Q_0^- \,\&\, Q_1^- \mid {\uparrow}P^+ \end{aligned}$$

Interesting. Call-By-Push-Value and the Calculus of Unity arose from completely different motivations, but still seem to make identical divisions between types: value types seem to be positive, and computation types seem negative. So the two foundations are talking about the same thing, right?

[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.

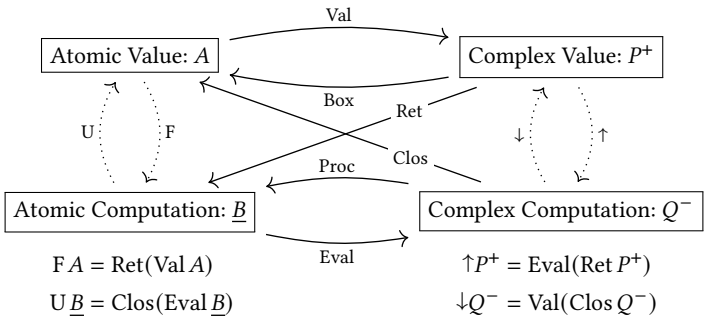

Fig. 1. The four kinds of types, and embeddings between them (dotted arrows are derived from solid ones).

Disagreements on who is first-class. The parallel between Call-By-Push-Value and focusing 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. Yet surprisingly, there is one glaring exception: they *disagree* on who has first-class status, able to be named and move freely. With Call-By-Push-Value, only values are first class. With focusing, interesting positive values are second class: they cannot be given *one* name, because the program can—and *must*—deconstruct them.

Our main idea is to combine these two similar systems while preserving their notions of second-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 [54]. 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 simple machine primitives like individual 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 (*e.g.,* we can always build bigger and bigger tuples), which is operationally *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 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 actual 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, which is signaled by Val. But to go the other way, a

complex data structure—which might bring together an arbitrary number of registers, memory, and a tag to describe the choice of constructor—cannot just be directly stuffed in a register. Instead, it has to be Boxed by storing its information and then using an atomic pointer to it instead.

The bottom is concerned with computations. A complex computation may need many immediate inputs stored in registers to run correctly, but an atomic computation just wants something simple like a pointer to the call stack, where more information could be retrieved as necessary. Eval punctuates the end of the input needed for a complex computation, evaluating to a single action that can be run. Proc *boxes* the calling context given to a complex computation—pushing it as a new frame onto the call stack—before running a simpler atomic action to decide what to do next.

The two diagonal arrows are the only ways to transition between values and computations. Ret describes computations which return a result. Ret begins with complex values so multiple results can be returned directly in registers, and ends up with an atomic computation that is ready to run. Likewise, Clos starts with a complex quadrant to build closures around code needing any amount of input (like a function), to give an atomic value represented as just a pointer.

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 $\downarrow$ and $\uparrow$ polarity shifts can be faithfully derived from the other ones, but not vice versa. Of course, $\mathrm{Clos}(\uparrow P^+)$ converts a complex value P^+ to an atomic one, but this is operationally very different from Box! Whereas Box P^+ is a pointer to where a fully-evaluated P^+ data structure is stored, $\mathrm{Clos}(\uparrow P^+)$ is a pointer to code that could generate and return the P^+ structure. 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.

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. 2. Being based on the λ-calculus, it is not as perfectly symmetric as something like the Calculus of Unity [54]; 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 $(1, \mathrm{val}\,\square, \mathrm{val}\,\square)[\mathrm{int}\,x, 3.14]$, we will write $(1, \mathrm{val}\,\mathrm{int}\,x, \mathrm{val}\,3.14)$. We also sometimes use syntactic sugar given in fig. 3 to write curried function applications in the more familiar λ-calculus style.

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 (most usually, a type) is named x and can be mentioned in s's type.

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:

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

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

$$\begin{array}{ll} \langle L\ S \parallel K \rangle = \langle L \parallel S \cdot K \rangle & \langle L\ b \parallel K \rangle = \langle L \parallel b \cdot K \rangle \\ \langle L\ V \parallel K \rangle = \langle L \parallel V \cdot K \rangle & L.\,\mathrm{eval}\,O = \langle L \parallel \mathrm{eval}\,O \rangle \end{array}$$

Fig. 3. Syntactic sugar for writing call stacks in functional style.

Actual concrete structures S are introduced by filling all $\Box$'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 $\Box$ 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 the bit b says which option should be called with k. There is also the polymorphic (*i.e.,* universal) specialization $\Box \cdot k$ of type $\forall R\,x : A.\,Q$ where the atomic value (*e.g.,* a type) placed in the $\Box$ is referred to as x in the type of k. The base case is $\mathrm{eval}\,O$, which serves as the final punctuation mark for a calling context that is finished, creating an atomic computation that is ready to run. The annotation O describes the context of that atomic computation: will it run as a sub-computation of the larger program (sub) that must return to some caller, or is it "naked" and running with no larger context at all (run).

Like with complex structures, concrete call stacks K are put together by filling the holes with values, written $k[V...]$, and taken apart by *copattern matching* [1]. Copatterns q are formed by filling a stack shape with distinct variables $k[R\ x : A...]$. *Copattern-matching code* F, *i.e.,* function code, is either a set of alternatives $\{\ q \to M...\ \}$ sending copatterns to an atomic computation M, or else 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 (box S of type Box P) or closure around function code (clos F of type 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 [21, 22] 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 $k[T..., V...]$. But to be sure, these type parameters should still be erasable because they never impact run-time behavior. 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 $R\ x$, to distinguish the different instructions like ref x for reading an address register named x or flt x for reading a floating-point register coincidentally named x as well.

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 **as** G matches the structure S against the patterns of G. But notice how trivial this expression is: S must already be a fully-built structure made with known constructors that have to exactly match against the patterns in G. In effect, every S **as** G expression can either be 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 **as** G. Instead, loading the contents of a boxed data structure is accomplished *exclusively* by **unbox** V **as** G, which immediately deconstructs its shape.

Of course, neither of these two computation forms is enough. We still need to sequence sub-computations and remember the results they return. In Call-By-Push-Value, this is done by a computation **do** $x \leftarrow M; M'$ which runs M until it returns a result named x before continuing to M'. Instead, Call-by-Unboxed-Value has a similar atomic computation **do** M **as** G with one key difference: the sub-computation returns M an *unboxed* result that is matched in place. The unboxed S returned by ret S is a second-class entity that cannot be named directly, since it might be comprised of *multiple values*, and can 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. Two options are directly invoking known code as λF, or calling a closure object as V. call. The final possibility is that the function code hasn't been figured out yet, 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 the form proc F which will pop that frame off the stack and continue as F.

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

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

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

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

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

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

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. 4. The Call-By-Unboxed-Value operational semantics.

3.2 Operational semantics

The Call-By-Unboxed-Value operational semantics are given in fig. 4. Despite the many, finely sub-divided syntactic categories, there are only eight reduction rules, many of which are 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:

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

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:

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

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:

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

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

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

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

3.3 Type system

The Call-By-Unboxed-Value type system is given in figs. 5 to 7. The various kinds of well-formed types are classified in fig. 5—all P and Q types are just complex with no further specification, written P : **cplx val** and Q : **cplx comp** respectively, but the atomic value types A are further separated by their representation, written $A : R$ **val**, and atomic computation types B are separated by the observational context, 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 represented as 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 complex and atomic values are classified in fig. 6. One of the primary 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 the structure that is built when those holes are

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

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. 5. The kinds of types and typing environments.

filled, and Γ is used to keep 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, $\Gamma \vdash V : A : R\ \mathbf{val}$, when their type has a known representation. Note that the premise of the *Match* rule is required to 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.

The types of complex and atomic computations are classified in fig. 7. Here, we have several 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 that can be called by the stack 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 the typed stack shape $k : Q$ to the left of the $\vdash$, similar to [11, 52], since these rules correspond to left rules of the sequent calculus. 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 what kind of observation is used to evaluate them. For certain atomic computations, like $\mathrm{ret}\ S : \mathrm{Ret}\ P : \mathrm{sub}\ \mathbf{comp}$ 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.

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 they encounter a generic type variable before

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. 6. Types of complex and atomic values.

reaching an atomic unit. For example, Val Float × Val Int describes only the shape (val □, val □) but ∃ ty a : Type **cplx val**. Val Float × a describes *no shapes*, since a generic ty a : **cplx val** has no known patterns. Not all polymorphism blocks pattern-matching, however, including even polymorphism over complex values. For example, both ∃ ty a : Type ref **val**. Val Float × Val a and ∃ ty a : Type **cplx val**. Val Float × Val(Box a) describe the same shape (□, val □, val □). The same scenario occurs in stack shapes, where Val Float → Eval Void describes (val □ · eval run), both types ∀ ty a : Type sub **comp**. Val Float → Eval a and ∀ ty a : Type **cplx comp**. Val Float → Eval(Proc a) describe the same shape (□ · val □ · eval sub), but ∀ ty a : Type **cplx comp**. Val Float → a describes *no shapes*. The second-class status of complex unboxed structures and call stacks automatically enforces the ad-hoc monomorphism restrictions imposed by [15, 19].

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 3.3 (Primitive Safety). $g : P$ implies that $\Gamma \vdash g(S)$: void : run **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.

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

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

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

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

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

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

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

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

Fig. 7. Types of complex and atomic computations.

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

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

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 $b : \mathrm{Type}\;\mathbf{cplx\;val}$: setting $b = \mathrm{Val\,Float} \times \mathrm{Val\,Int}$ "returns" an unboxed integer-float pair, setting $b = \mathrm{Val}(\mathrm{Box}\,P)$ "returns" a reference to a fully-evaluated boxed structure, and $b = \mathrm{Val}(\mathrm{Clos}\,Q)$ "returns" a reference to unevaluated code. 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.

With the assumption that all primitive operations are safe, we can prove the usual type safety property for the Call-By-Unboxed-Value λ-calculus (see appendix B for proofs).

Lemma 3.4 (Progress). *If* $\bullet \vdash M : \mathrm{void} : \mathrm{run}\;\mathbf{comp}$*, then either* $M \mapsto M'$ *or* M terminal.

Lemma 3.5 (Preservation). *If* $\Gamma \vdash M : B : O\;\mathbf{comp}$ *and* $M \mapsto M'$ *then* $\Gamma \vdash M' : B : O\;\mathbf{comp}$.

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

Fig. 8. Extensional η axioms of the typed equational theory.

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. 8. 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. 4 and 8).

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:

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

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

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

Similarly, we can 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:

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

These equations are derivable from the $\beta\eta$ axioms seen already in figs. 4 and 8 (see appendix C).

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:

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

so that calling $\langle \lambda\, idFlt \parallel \mathrm{val}\,3.14 \cdot \mathrm{eval\,sub}\rangle$ successfully matches the copattern, which computes to $\mathrm{ret\,val}\,3.14$, 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:

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

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 $\mathrm{Clos}(\ldots)$ 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 $\mathrm{clos}\,\{\ldots\} : \mathrm{Clos}(\mathrm{Eval}(\mathrm{Ret}(\mathrm{Val\,Int})))$ 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:

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

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,[2] like so:

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

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 $(\mathrm{Bool} \times \mathrm{Bool}) \to \mathrm{Eval}(\mathrm{Ret\,Bool})$ rearranges the parentheses slightly, but essentially corresponds to the same low-level code. What may be more

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

surprising is that merging the two booleans together into another sum type Bool + 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):

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

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 [30, 55], like so:

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

where we now start using familiar functional-style application from fig. 3. 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,

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

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

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

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

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

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

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

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 $\mathit{maybeAdd} : \mathrm{Val}(\mathrm{Box}(1 + \mathrm{Val\,Nat})) \to \mathrm{Val\,Nat} \to \uparrow \mathrm{Nat}$; 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 $maybeAdd : \downarrow\uparrow(1+\mathrm{Val\,Nat}) \to \downarrow\uparrow\mathrm{Nat} \to \uparrow\mathrm{Nat}$, 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:

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

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

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

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

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

Here, we can assume that $f.\mathrm{call}$ 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 $\mathrm{val\,ref}\,x \cdot \mathrm{val\,ref}\,x \cdot \mathrm{eval\,sub}$ 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

$$\mathbf{class}\ \mathrm{Num}\,a\ \mathbf{where}\ \ (+) \quad :: a \to a \to a$$
$$negate :: a \to a$$

introduces overloaded operators $(+) :: \mathrm{Num}\,a \Rightarrow a \to a \to a$ and $negate :: \mathrm{Num}\,a \Rightarrow a \to a \to a$ that work for any instance of $\mathrm{Num}\,a$. 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:

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

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 $\mathit{negate} : \forall\, \mathrm{ty}\, a : \mathrm{Type}\ \mathbf{cplx\ val}.\ \mathrm{Num}\, a \to a \to \uparrow a$ because there is no pattern for an unknown $\mathrm{ty}\, a : \mathbf{cplx\ val}$. However, we *can* easily implement these functions which merely extract and return one of the closures in the dictionary.

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

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:

$$\mathit{NumInt} : \mathrm{Num\ Int} \qquad \mathit{NumFlt} : \mathrm{Num\ Float}$$
$$\mathit{NumInt} = (\mathrm{clos}\ \mathit{add\#}, \mathrm{clos}\ \mathit{negate\#}) \qquad \mathit{NumFlt} = (\mathrm{clos}\ \mathit{addFlt\#}, \mathrm{clos}\ \mathit{negateFlt\#})$$

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 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.[3]

We can then embed this Polymorphic Call-By-Push-Value λ-calculus into Call-By-Unboxed-Value as shown in fig. 9. The key idea of this embedding is to interpret the (potentially polymorphic) types with uniform representations. Every value type $A : \mathbf{val}$ is interpreted as an atomic reference $[\![A]\!] : \mathrm{ref}\ \mathbf{val}$, and every computation type $\underline{B} : \mathbf{comp}$ is interpreted as an atomic subroutine $[\![\underline{B}]\!] : \mathrm{sub}\ \mathbf{comp}$. This uniform representation, unsurprisingly, forces us to insert boxes around every complex value (tuples, sum types, and packages). On the computation side, we force every computation to be a simple subroutine via sub-procedures ($\mathrm{proc}\,\{\ldots\}$) that we can just evaluate. Note that the examples seen thus far (in section 4) hadn't made much use of the Proc Q type, 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 (see appendix D for proofs).

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.*

[3] For the full formal definition, see appendix D.

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

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

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

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

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

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

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

Fig. 9. The translation from Polymorphic Call-By-Push-Value to Call-By-Unboxed-Value.

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. $CBPV[\![M]\!] = CBPV[\![M']\!]$.*

6 AN UNBOXED ABSTRACT MACHINE

Having studied Call-By-Unboxed-Value from the high-level—as a suitable target for semantics-preserving compilation of standard functional programs—we need to also consider it from a lower-level vantage point, to be sure that it can actually be implemented with the intended memory behavior on realistic machines. In particular, the *only* objects that get written to long-term storage are reference values box S and clos F, as well as subroutine stack frames corresponding to evaluation contexts $\mathbf{do}\ \square\ \mathbf{as}\ G$ and $\langle \square.\ \mathrm{enter} \parallel K \rangle$, which are stored contiguously at one address—everything else can be held in a simpler but more accessible 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 main piece of information

that is needed to compile the code is about closure environments: when code is written into long-term storage, we need to know what are the relevant variables to copy into the closure, and thus how they are represented. We can extend 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, this extra 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:

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

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 type assignments of the free variables actually found in F (*i.e.,* $FV(F)$). As shorthand for inspecting copatterns, we write $k[\Gamma, O]$ to mean that k ends in eval O. The rest of the cases just proceed directly by recursion. 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 $AM[\![M.\,\mathrm{enter}]\!]_\Gamma = AM[\![M]\!]^{\mathrm{sub}}_\Gamma.\,\mathrm{eval}$.

6.2 Machine configurations and transitions

The definition of the abstract machine is given in fig. 10. Notice that the machine configuration m is a combination of three parts: a command c that says what to do, local information about registers ρ and κ, and long-term storage σ. Each ρ register is fixed to a certain 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 representation as well as its name. Note that 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: the x addresses are heap-allocated and need to be garbage collected, because they live for an unpredictable amount of time, but $\overline{x}$ addresses follow a linear stack discipline, and can be allocated and freed as a traditional, contiguous call stack. Code in both kinds of stored objects, clos $F[\rho]$ and **do** $G[\rho\kappa]$, is closed over the contents of (value and stack) registers at the time they were stored.

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 bound by 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

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

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

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 \mathrm{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}$$

$$\begin{array}{rl}
(\mathit{Fun}) & \langle \{\, k[\Gamma_k] \to M_k \overset{k \in Q}{\ldots} \parallel k' \parallel W\ldots \,\}\rangle\ [\rho\kappa][\sigma] \mapsto \langle M_{k'}\rangle\ [\rho, (\Gamma_{k'} := W\ldots)\kappa][\sigma] \\
(\mathit{Match}) & \langle s' \parallel W\ldots \parallel \{\, s[\Gamma_s] \to M_s \overset{s \in P}{\ldots} \,\}\rangle\ [\rho\kappa][\sigma] \mapsto \langle M_{s'}\rangle\ [\rho, (\Gamma_{s'} := W\ldots)\kappa][\sigma]
\end{array}$$

$$\begin{array}{rl}
(\mathit{Prim}_f) & \langle f \parallel k \parallel W\ldots\rangle\ [\rho\kappa][\sigma] \mapsto \langle \mathit{Prim}_f(k[V\ldots])\rangle\ [\rho'\kappa][\sigma] \quad (\text{if } V\ldots; \rho' = \mathit{Load}_f(k, W\ldots)) \\
(\mathit{Prim}_g) & \langle s \parallel W\ldots \parallel g\rangle\ [\rho\kappa][\sigma] \mapsto \langle \mathit{Prim}_g(s[V\ldots])\rangle\ [\rho'\kappa][\sigma] \quad (\text{if } V\ldots; \rho' = \mathit{Load}_g(s, W\ldots))
\end{array}$$

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. 10. 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.

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 $\{\, k[\Gamma] \to M\ldots \,\}$

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

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

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

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

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

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

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

Registers $\boxed{AM[\![\rho]\!]^{-1} = W/R\ x...}$, heap objects $\boxed{AM[\![H]\!]_\sigma^{-1} = V}$ 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. 11. Decompilation of the abstract machine.

or $\{\ s[\Gamma] \to M...\ \}$, then the *Fun* and *Match* rules do a switch statement on the shape s or k, and then bind the associated values $W...$ 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 $Load_f(k, W...) = V...; \rho$ and $Load_g(s, W...) = V...; \rho$ that abstracts out (at least) all the in $W...$ 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: both structure and stack shapes are devoid of any information about the atomic values that are held "inside," and don't even assign some name to each position, leaving them blank as just $\square$. The only run-time requirement we have for the shapes of complex types is that they are all distinct. As such, each shape for a complex type can be boiled down to just a single constant, *i.e.,* a sufficient number of bits to distinguish each possibility, so that choosing a branch is just a switch statement on a constant. Once the branch has been selected, the associated atomic values $W...$ can be assigned their local names to evaluate the next computation.

6.3 Back-translation and bisimulation

The abstract machine in fig. 10 is meant to correctly implement the low-level details left abstract in operational semantics in fig. 4. 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.

Decompiling commands and configurations, as shown in fig. 11, 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 (see appendix E for proofs).

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 $M = AM[\![m]\!]^{-1}$

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

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 and safety

Decompilation does more than relate the high- and low-level dynamic semantics; it also forms a link between the 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. This means well-typed programs correspond to well-typed abstract machine configurations—following the typing rules given in fig. 12—with its own type safety property. The key to typing the machine is to realize that there are two levels of environments corresponding to the two closing $\rho\kappa$ 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 (See appendix F for proofs).

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

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.

7 FUTURE AND RELATED WORK

Optimizing unboxed data and curried functions. Call-By-Unboxed-Value follows in the tradition of [46] in modeling the boxed versus unboxed status of a value as a feature in a compiler's intermediate language. This idea was extended to allow for polymorphism over a value's representation [19], as well as the calling convention of a function [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 examples of programs that abstract over types with different representations (section 4). Still, there may yet be some other applications that want the ability to abstract over representations or observations, which we leave to future work.

Updated typing rules for annotated closures and do-sequences:

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

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

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. 12. The Call-By-Unboxed-Value abstract machine type system.

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, 29, 34]. Usually, the question of how many arguments a function "really" requires (*i.e.,* it's *arity*) is an informal property that requires some complex compile-time analysis [9, 47, 53] and can be easily changed by program optimizations [23]. 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, 27]. More recent approaches to typed closure conversion [2, 37] represent them abstractly [8], which has also been modeled in a Call-By-Push-Value framework [49].

Adjoint calculi. Call-By-Unboxed-Value is very explicitly inspired by previous work on *adjoint calculi* [30, 31, 41, 42, 54, 55]. In essence, these calculi are similar to the more familiar monadic framework of computation [40], 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 adjunctive pair of functors. Furthermore, these calculi are famous for their ability to accurately express the semantics of types, such as "strong sums" [43], especially in the presence of side effects [32], making good on the promise that even effectful programs have the expected isomorphisms between types [33] 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. However, in practice, non-strict functional languages use *call-by-need* [6, 7] evaluation to *memoize* (*i.e.,* remember) answers and avoid re-computing them over and over. As such, we cannot simply "evaluate" a memoized computation and grab the result; somewhere the answer must be recorded for efficient retrieval in the future. Previous work has shown how to extend Call-By-Push-Value with call-by-need, but at the cost of losing the extensional η equalities [36] or with 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 context of a *memoizing computation* (memo **comp**) is always represented as *two* references memo $x, \overline{x}$: one ($\overline{x}$) to the stack frame that wants the answer, as usual, and another (x) to the thunk itself to be overwritten with the answer. More concretely, we could add memoizing computations with *tagless* [45] thunks to Call-By-Unboxed-Value abstract machine like so (optionally with closure annotations Γ'):

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

$$\rho^*(V\ldots, \mathrm{start}\, M[\Gamma]) = W\ldots, \mathrm{ref}\, x; x := \mathrm{start}\, M[\rho|_\Gamma] \qquad (\rho^*(V\ldots) = W\ldots; \sigma) \quad (\sigma(\rho(\mathrm{ref}\, x)) = \mathrm{start}\, M[\rho'])$$

$$\langle x.\mathrm{play}\rangle\ [\rho\ \mathrm{sub}\ \overline{x}]\,[\sigma] \mapsto \langle M\rangle\ [\rho'\ \mathrm{memo}\ x, \overline{x}]\,[\sigma]$$

$$\langle \mathrm{pause}\, M[\Gamma]\rangle\ [\rho\ \mathrm{memo}\ x, \overline{x}]\,[\sigma] \mapsto \langle M\rangle\ [\rho\ \mathrm{sub}\ \overline{x}]\,[\sigma, x := \mathrm{start}(\mathrm{pause}\, M[\Gamma])\,[\rho|_\Gamma]]$$

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, and 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 effectively 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 contrast to swapping pairs $P_0 \times P_1 \not\approx P_1 \times P_0$ that can change their 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 [50] to give an explicit mechanism for adding 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 $P_0 + P_1 \not\approx P_1 + P_0$ separate.

Join points. Sharing code is an important concern in practical implementations, especially when the representation forces the program to repeat similar work over multiple possible branches [26]. 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, 25].

Another way to share code is with *join points* [35] 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 [54] 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, 52], 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 GHC's Core representation, for similar reasons. Once we have done so, it is simpler to formalize the 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. 5).

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 [24, 44]. Despite this, there have been some promising starts based on Call-By-Push-Value [44, 51] and sequent calculi [38, 39, 48]. 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 material is based on work supported by the National Science Foundation under Grant No. 2245516.

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] Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph. D. Dissertation. Université Paris 7.
[22] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types.* Cambridge University Press, New York, NY, USA.
[23] 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.
[24] 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.
[25] 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
[26] 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
[27] Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320.
[28] Olivier Laurent. 2002. *Étude de la polarisation en logique.* Ph. D. Dissertation. Université de la Méditerranée - Aix-Marseille II.
[29] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language.* Technical report 117. INRIA.
[30] Paul Blain Levy. 2001. *Call-By-Push-Value.* Ph. D. Dissertation. Queen Mary and Westfield College, University of London.
[31] 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
[32] Paul Blain Levy. 2006. *Jumbo λ-Calculus.* Springer Berlin Heidelberg, Berlin, Heidelberg, 444–455. https://doi.org/10.1007/11787006_38
[33] 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
[34] 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.
[35] 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.
[36] 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.
[37] 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.
[38] Étienne Miquey. 2017. *Classical realizability and side-effects. (Réalisabilité classique et effets de bords).* Ph. D. Dissertation. University of the Republic, Montevideo, Uruguay.
[39] 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
[40] 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
[41] 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.
[42] Guillaume Munch-Maccagnoni. 2013. *Syntax and Models of a non-Associative Composition of Programs and Proofs.* Ph. D. Dissertation. Université Paris Diderot.

[43] 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.
[44] 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
[45] 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.
[46] 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.
[47] 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.
[48] Arnaud Spiwack. 2014. A Dissection of L. https://assert-false.science/arnaud/papers/A%20dissection%20of%20L.pdf
[49] 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
[50] 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.
[51] Matthijs Vákár. 2017. *In Search of Effectful Dependent Types*. Ph. D. Dissertation. Magdalen College, University of Oxford.
[52] 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
[53] Dana N. Xu and Simon L. Peyton Jones. 2005. Arity Analysis. (2005). Working notes.
[54] Noam Zeilberger. 2008. On the Unity of Duality. *Annals of Pure and Applied Logic* 153, 1 (2008), 660–96.
[55] Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching*. Ph. D. Dissertation. Carnegie Mellon University.

Contents

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 each case. 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 [35], we can easily add some syntactic sugar for letting us assign a name to a whole complex value, corresponding to Zeilberger's [55] *complex variables*.

However, [55]'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. 13. 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 $x \in \{ s_i^{i \in I} \}$, where the set $\{ s_i^{i \in I} \}$ disambiguates all the possible different shapes that the complex x might take.

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

$$\begin{array}{rl} \mathit{PatternCxt} \ni p^1 & ::= \square \mid p^1, p \mid p, p^1 \mid b, p^1 \mid R\ x : T, p^1 \\ \mathit{CoPatternCxt} \ni q^1 & ::= p^1 \cdot q \mid p \cdot q^1 \mid b \cdot q^1 \mid R\ x : A \cdot q^1 \\ \mathit{StackPrefix} \ni k^{pre} & ::= \square \mid s \cdot k^{pre} \mid b \cdot k^{pre} \mid \square \cdot k^{pre} \end{array}$$

$$\{\ldots; p^1[x \in \{\, s_i \overset{i \in I}{\ldots} \,\}] \to M\ \} \to \{\ldots; p^1[s_i[x_i\ldots]] \to M[s_i[x_i\ldots]/x] \overset{i \in I}{\ldots}\ \}$$

$$\{\ldots; q^1[x \in \{\, s_i \overset{i \in I}{\ldots} \,\}] \to M\ \} \to \{\ldots; q^1[s_i[x_i\ldots]] \to M[s_i[x_i\ldots]/x] \overset{i \in I}{\ldots}\ \}$$

$$\{\ldots; k^{pre}[x\ldots, \mathrm{more} \in \{\, k'_i \overset{i \in I}{\ldots} \,\}] \to M\ \} \to \{\ldots; k^{pre}[x\ldots, k'_i[y_i\ldots]] \to \langle M \parallel k'_i[y_i\ldots] \rangle \overset{i \in I}{\ldots}\ \}$$

$$\langle S\ \mathbf{as}\ \{\, p \to M_p \overset{p \in P}{\ldots} \,\} \parallel K \rangle \to S\ \mathbf{as}\ \{\, p \to \langle M_p \parallel K \rangle \overset{p \in P}{\ldots} \,\}$$

$$\langle \mathbf{unbox}\ V\ \mathbf{as}\ \{\, p \to M_p \overset{p \in P}{\ldots} \,\} \parallel K \rangle \to \mathbf{unbox}\ V\ \mathbf{as}\ \{\, p \to \langle M_p \parallel K \rangle \overset{p \in P}{\ldots} \,\}$$

$$\langle \mathbf{do}\ M\ \mathbf{as}\ \{\, p \to M_p \overset{p \in P}{\ldots} \,\} \parallel K \rangle \to \mathbf{do}\ M\ \mathbf{as}\ \{\, p \to \langle M_p \parallel K \rangle \overset{p \in P}{\ldots} \,\}$$

$$s[\delta] \in \{\, s_i \overset{i \in I}{\ldots} \,\} \to s[\delta] \qquad (s \in \{\, s_i \overset{i \in I}{\ldots} \,\})$$

$$\langle \langle L \parallel k^{pre}[\delta, \mathrm{more} \in \{\, k_i \overset{i \in I}{\ldots} \,\}] \rangle \parallel k'[\delta'] \rangle \to \langle L \parallel k^{pre}[\delta, k'[\delta']] \rangle \qquad (k' \in \{\, k_i \overset{i \in I}{\ldots} \,\})$$

Fig. 14. Untyped macro expansion of complex (co)pattern disjunction

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 with $\mathrm{more} \in \{\, k_i \overset{i \in I}{\ldots} \,\}$, 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. 2. This elaboration is shown in fig. 14. 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 [55]. Moreover, we did not need to complicate the language of representations or observations to 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:

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

Desugaring this definition using fig. 14 gives exactly the fully-elaborated, four-way branching version from section 4.

B CALL-BY-UNBOXED-VALUE TYPE SAFETY

Lemma B.1 (Well-Typed Substitution). *Given* $\Gamma \vdash V : A : R\,\mathbf{val}$, *then*

(1) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \vdash T : \tau$ *implies* $\Gamma, \Gamma'[V/R\,x] \vdash T[V/R\,x] : \tau$,
(2) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \vdash V' : A' : R'\,\mathbf{val}$ *implies* $\Gamma, \Gamma'[V/R\,x] \vdash V'[V/R\,x] : A'[V/R\,x] : R'\,\mathbf{val}$,
(3) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \vdash M : B : O\,\mathbf{comp}$ *implies* $\Gamma, \Gamma'[V/R\,x] \vdash M[V/R\,x] : \Phi[V/R\,x]$,
(4) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \vdash L : Q$ *implies* $\Gamma, \Gamma'[V/R\,x] \vdash L[V/R\,x] : Q[V/R\,x]$,
(5) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \vdash S : P$ *implies* $\Gamma, \Gamma'[V/R\,x] \vdash S[V/R\,x] : P[V/R\,x]$,
(6) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \mid K : Q \vdash \Phi$ *implies* $\Gamma, \Gamma'[V/R\,x] \mid K[V/R\,x] : Q[V/R\,x] \vdash \Phi[V/R\,x]$,
(7) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \vdash F : Q$ *implies* $\Gamma, \Gamma[V/R\,x] \vdash F[V/R\,x] : Q[V/R\,x]$,
(8) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \mid G : P \vdash \Phi$ *implies* $\Gamma, \Gamma[V/R\,x] \mid G[V/R\,x] : P[V/R\,x] \vdash \Phi$, *and*
(9) $\Gamma, x : A : R\,\mathbf{val}, \Gamma' \vdash V' \ldots : \Delta$ *implies* $\Gamma, \Gamma'[V/R\,x] \vdash V'[V/R\,x] \ldots : \Delta[V/R\,x]$.

Proof (sketch). By induction on the typing derivation of the expression being substituted into. In the base case, a *Var* or *TyVar* axiom for the variable being substituted for is replaced by the given typing derivation of the atomic value that replaces it. The rest of the cases follow directly from the inductive hypothesis. The reason that the substitution may be seen in the types to the right of the replaced variable (in Γ', Φ, B, P, *etc.*) is in the special case where $R = \mathrm{ty}$ and $V = T$ is itself a type that may be referenced elsewhere in the judgment. In all other cases for non-type values (represented as int, flt, and ref), the free x cannot appear in any types, so the only substitution that is relevant is the main term. □

Lemma B.2 (Open Progress). *If* $\Gamma \vdash M : B : O\,\mathbf{comp}$, *then* $M \mapsto M'$ *or* M *is one of the following:*

(1) $M = \mathrm{ret}\, S$ *or* $M = \mathrm{proc}\, F$, *only when* $O = \mathrm{sub}$,
(2) $M = E[\mathbf{unbox}\, x\, \mathbf{as}\, G]$ *or* $M = E[\langle x.\, \mathrm{call} \,\|K\rangle]$, *for some free variable* x, *or*
(3) M terminal.

Proof (sketch). By induction on the typing derivation of $\Gamma \vdash M : B : O\,\mathbf{comp}$. The proof is standard, following the inductive hypothesis to reduce sub-computations, and using the fact that well-typed substitution preserves typing derivations (lemma B.1).

Of note, in S **as** G pattern matching always succeeds or is terminal because S definitionally has the form $s[V\ldots]$ and the typing rules *Match*, *PrimMatch*, and *StructCut* ensure one of G's patterns has the shape s, and that the number and representations of $V\ldots$ matches the variables bound by that pattern. Similarly, $\langle \lambda F \parallel K \rangle$ always takes a step or is terminal because the typing rules *CoMatch*, *PrimFun*, and *StackCut* force F to correctly cover the case $K = k[V\ldots]$. □

Lemma 3.4 (Progress). *If* $\bullet \vdash M : \text{void} : \text{run}\ \mathbf{comp}$*, then either* $M \mapsto M'$ *or* M terminal.

Proof. Follows from lemma B.2, since the assumption $\bullet \vdash M : \text{void} : \text{run}\ \mathbf{comp}$ rules out the other possibilities because (1) run ≠ sub and (2) M cannot refer to free variables. □

Lemma 3.5 (Preservation). *If* $\Gamma \vdash M : B : O\ \mathbf{comp}$ *and* $M \mapsto M'$ *then* $\Gamma \vdash M' : B : O\ \mathbf{comp}$.

Proof (sketch). By induction on the typing derivation of $\Gamma \vdash M : B : O\ \mathbf{comp}$ and on the evaluation context of the reduction $E[M_1] \mapsto E[M_1']$, where M_1 is the sub-term where the reduction rule is applied, using the property that well-typed substitution preserves typing (lemma B.1). □

C DERIVATION OF COMMUTING CONVERSION EQUALITIES

- To commute proc with **do**, note that, for any copattern q, the reassociated $Assoc(q)[\Box.\ \text{enter}]$ is an evaluation context, so it can be commuted with the **do**-statement (via *cc* Ret) like so:

$$\begin{aligned}
&\mathbf{do}\ M\ \mathbf{as}\ \{\ p \to \text{proc}\ \{\ q \to M'_{qp}{}^{q\in Q}_{\ldots}\ \}^{p\in P}_{\ldots}\ \} \\
&=_{\eta\,\text{Proc}}\ \text{proc}\ \{\ q \to Assoc(q)[\mathbf{do}\ M\ \mathbf{as}\ \{\ p \to \text{proc}\ \{\ q \to M'_{qp}{}^{q\in Q}_{\ldots}\ \}^{p\in P}_{\ldots}\ \}.\ \text{enter}]^{q\in Q}_{\ldots}\ \} \\
&=_{cc\,\text{Ret}}\ \text{proc}\ \{\ q \to \mathbf{do}\ M\ \mathbf{as}\ \{\ p \to Assoc(q)[\text{proc}\ \{\ q \to M'_{qp}{}^{q\in Q}_{\ldots}\ \}.\ \text{enter}]^{p\in P}_{\ldots}\ \}^{q\in Q}_{\ldots}\ \} \\
&=_{\beta\,\text{Proc}}\ \text{proc}\ \{\ q \to \mathbf{do}\ M\ \mathbf{as}\ \{\ p \to Assoc(q)[\lambda\ \{\ q \to M'_{qp}{}^{q\in Q}_{\ldots}\ \}]^{p\in P}_{\ldots}\ \}^{q\in Q}_{\ldots}\ \} \\
&=_{\beta\lambda}\ \text{proc}\ \{\ q \to \mathbf{do}\ M\ \mathbf{as}\ \{\ p \to M'_{qp}{}^{p\in P}_{\ldots}\ \}^{q\in Q}_{\ldots}\ \}
\end{aligned}$$

- To commute proc with **unbox**, we can immediately apply η Box to move the **unbox** across arbitrary contexts, including the proc, like so:

$$\begin{aligned}
&\text{proc}\ \{\ q \to \mathbf{unbox}\ V\ \mathbf{as}\ \{\ p \to M_{qp}{}^{p\in P}_{\ldots}\ \}^{q\in Q}_{\ldots}\ \} \\
&=_{\eta\,\text{Box}}\ \mathbf{unbox}\ V\ \mathbf{as}\ \{\ p \to \text{proc}\ \{\ q \to \mathbf{unbox}\ \text{box}\ p\ \mathbf{as}\ \{\ p \to M_{qp}{}^{p\in P}_{\ldots}\ \}^{q\in Q}_{\ldots}\ \}^{p\in P}_{\ldots}\ \} \\
&=_{\beta\,\text{Box}}\ \mathbf{unbox}\ V\ \mathbf{as}\ \{\ p \to \text{proc}\ \{\ q \to p\ \mathbf{as}\ \{\ p \to M_{qp}{}^{p\in P}_{\ldots}\ \}^{q\in Q}_{\ldots}\ \}^{p\in P}_{\ldots}\ \} \\
&=_{\beta\,\mathbf{as}}\ \mathbf{unbox}\ V\ \mathbf{as}\ \{\ p \to \text{proc}\ \{\ q \to M_{qp}{}^{q\in Q}_{\ldots}\ \}^{p\in P}_{\ldots}\ \}
\end{aligned}$$

- To commute proc with a bare **as**-match, note that the structure of the syntax forces *all* well-typed structures (even open ones) to decompose into a shape of their complex value type filled with appropriately-typed values, so a β **as** rule will always apply, like so:

$$\begin{aligned}
&S\ \mathbf{as}\ \{\ p \to \text{proc}\ \{\ q \to M_{qp}{}^{q\in Q}_{\ldots}\ \}^{p\in P}_{\ldots}\ \} \\
&= s'[V\ldots]\ \mathbf{as}\ \{\ s[R\ x : A\ldots] \to \text{proc}\ \{\ q \to M_{qs}{}^{q\in Q}_{\ldots}\ \}^{s\in P}_{\ldots}\ \} \\
&=_{\beta\,\mathbf{as}}\ \text{proc}\ \{\ q \to M_{qs'}[V/R\ x\ldots]^{q\in Q}_{\ldots}\ \} \\
&=_{\beta\,\mathbf{as}}\ \text{proc}\ \{\ q \to s'[V\ldots]\ \mathbf{as}\ \{\ s[R\ x : A\ldots] \to M_{qs}{}^{s\in P}_{\ldots}\ \}^{q\in Q}_{\ldots}\ \} \\
&= \text{proc}\ \{\ q \to S\ \mathbf{as}\ \{\ p \to M_{qp}{}^{p\in P}_{\ldots}\ \}^{q\in Q}_{\ldots}\ \}
\end{aligned}$$

D PROOF OF CALL-BY-PUSH-VALUE'S TYPE AND EQUATIONAL CORRESPONDENCE

The full definition of the polymorphic Call-By-Push-Value λ-calculus is given in figs. 15 to 18.

Based on this definition, we can prove the that the embedding of Call-By-Push-Value into to Call-By-Unboxed-Value is sound with respect to typing and equality.

$$
\begin{array}{rl}
\mathit{Kind} \ni \tau & ::= \mathbf{val} \mid \mathbf{comp} \\
\mathit{Type} \ni T & ::= A \mid \underline{B} \\
\mathit{ValueType} \ni A & ::= X \mid 1 \mid A_0 \times A_1 \mid 0 \mid A_0 + A_1 \mid \exists X : \tau . A \mid \mathrm{U}\,\underline{B} \\
\mathit{CompType} \ni \underline{B} & ::= X \mid A \to \underline{B} \mid \top \mid \underline{B}_0 \,\&\, \underline{B}_1 \mid \forall X : \tau . \underline{B} \mid \mathrm{F}\,A \\
\\
\mathit{Value} \ni V & ::= x \mid () \mid (V_0, V_1) \mid (\mathtt{0}, V) \mid (\mathtt{1}, V) \mid \mathrm{thunk}\, M \\
\mathit{Comp} \ni M & ::= \mathbf{do}\, x : A \leftarrow M \,\mathbf{in}\, M' \mid \mathrm{return}\, V \mid V.\,\mathrm{force} \\
& \quad \mid \mathbf{match}\, V \,\mathbf{as}() \to M \mid \mathbf{match}\, V \,\mathbf{as}(x_0 : A_0, x_1 : A_1) \to M \\
& \quad \mid \mathbf{match}\, V \,\mathbf{as}\, \{\, (b, x_b : A_b) \to M_b{}^{b \in \{0,1\}}_{\cdots} \,\} \mid \mathbf{match}\, V \,\mathbf{as}(X : \tau, x : A) \to M \\
& \quad \mid \lambda x{:}A.M \mid M\ V \mid \lambda\, \{\ \} \mid \lambda\, \{\, b.M^{b \in \{0,1\}}_{\cdots} \,\} \mid M\ \mathtt{0} \mid M\ \mathtt{1} \mid \Lambda X{:}\tau.M \mid M\ T
\end{array}
$$

Fig. 15. Polymorphic Call-By-Push-Value syntax.

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

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

Fig. 16. Polymorphic Call-By-Push-Value operational semantics.

Lemma D.1 (Type Soundness). *Let $[\![\_]\!]$ denote CBPV$[\![_]\!]$ in the following:*
(1) If $\Gamma \vdash T : \tau$ in Polymorphic CBPV then $[\![\Gamma]\!] \vdash [\![A]\!] : [\![\tau]\!]$ in CBUV.
*(2) If $\Gamma \vdash V : A$ in Polymorphic CBPV then $[\![\Gamma]\!] \vdash [\![V]\!] : [\![A]\!] :$ ref **val** in CBUV.*
*(3) If $\Gamma \vdash M : \underline{B}$ in Polymorphic CBPV then $[\![\Gamma]\!] \vdash [\![M]\!] : [\![\underline{B}]\!] :$ sub **comp** in CBUV.*

Proof (sketch). By (mutual) induction on the typing derivations of $\Gamma \vdash T : \tau$ and $\Gamma \vdash V : A$ and $\Gamma \vdash M : \underline{B}$. □

Lemma D.2 (Equational Soundness). *Polymorphic Call-By-Push-Value's equational theory is sound with respect to Call-By-Unboxed-Value: if $M =_{\beta\eta cc} M'$ then CBPV$[\![M]\!] =_{\beta\eta}$ CBPV$[\![M']\!]$, and similarly for values and types.*

Proof. Let $[\![\_]\!]$ denote *CBPV*$[\![\_]\!]$ in the following.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 17. Polymorphic Call-By-Push-Value type system.

Soundness of β reduction rules:

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:

$$(\eta\to) \quad \lambda x{:}A.(M\ x) = M \quad : A \to \underline{B} \quad (x \notin FV(M))$$

$$(\eta 0) \quad \lambda\,\{\ \} = M \quad : \top$$

$$(\eta\&) \quad \lambda\,\{\ \mathtt{0}.(M\ \mathtt{0}); \mathtt{1}.(M\ \mathtt{1})\ \} = M \quad : \underline{B}_0\ \&\ \underline{B}_1$$

$$(\eta\forall) \quad \Lambda X{:}\tau.(M\ X) = M \quad : \forall X{:}\tau.\underline{B} \quad (X \notin FV(M))$$

$$(\eta\,\mathrm{U}) \quad \mathrm{thunk}(M.\,\mathrm{force}) = M \quad : \mathrm{U}\,\underline{B}$$

$$(\eta\,\mathrm{F}) \quad \mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\ \mathrm{return}\ x = M \quad : \mathrm{F}\,A$$

$$(\eta 1) \quad \mathbf{match}\ V\ \mathbf{as}\ \{\ () \to M[()/x]\ \} = M[V/x] \quad (V : 1)$$

$$(\eta\times) \quad \mathbf{match}\ V\ \mathbf{as}(x_0{:}A_0, x_1{:}A_1) \to M[(x_0,x_1)/x] = M[V/x] \quad (V : A_0 \times A_1,\quad x_0, x_1 \notin FV(M))$$

$$(\eta 0) \quad \mathbf{match}\ V\ \mathbf{as}\ \{\ \} = M[V/x] \quad (V : 0)$$

$$(\eta+) \quad \mathbf{match}\ V\ \mathbf{as}\{(b, x_b{:}A_b) \to M_b[(b,x_b)/x]^{b\in\{0,1\}}\} = M[V/x] \quad (V : A_0 + B_0,\quad x_b \notin FV(M_b))$$

$$(\eta\times) \quad \mathbf{match}\ V\ \mathbf{as}(X{:}\tau, x{:}A) \to M[(X,x)/x] = M[V/x] \quad (V : \exists X{:}\tau.A,\quad X, x \notin FV(M))$$

Sequencing axioms:

$$(cc\to) \quad \begin{array}{l}\mathbf{do}\ x : A \leftarrow M\\ \mathbf{in}\ (\lambda y : A'.M')\end{array} = \lambda y : A'.(\mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\ M') \quad (y \notin FV(M))$$

$$(cc\&) \quad \begin{array}{l}\mathbf{do}\ x : A \leftarrow M\\ \mathbf{in}\ (\lambda\,\{\ b.\ M_b{}^{b\in\{0,1\}}\ \})\end{array} = \lambda\,\{\ b.(\mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\ M_b)^{b\in\{0,1\}}\ \}$$

$$(cc\forall) \quad \begin{array}{l}\mathbf{do}\ x : A \leftarrow M\\ \mathbf{in}\ (\Lambda X : \tau.M')\end{array} = \Lambda X : \tau.(\mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\ M') \quad (X \notin FV(M))$$

$$(cc\,\mathrm{F}) \quad \begin{array}{l}\mathbf{do}\ x : A \leftarrow M\ \mathbf{in}\\ \mathbf{do}\ y : A' \leftarrow M'\\ \mathbf{in}\ M''\end{array} = \begin{array}{l}\mathbf{do}\ x : A \leftarrow (\ \mathbf{do}\ y : A \leftarrow M'\\ \qquad\qquad \mathbf{in}\ M)\\ \mathbf{in}\ M''\end{array} \quad (x \notin FV(M'), y \notin FV(M))$$

Fig. 18. Polymorphic Call-By-Push-Value equational theory.

$$(\beta 1)\ \ \mathbf{match}\ ()\ \mathbf{as}\ \{\ () \to M\ \} = M$$

$$[\![\mathbf{match}\ ()\ \mathbf{as}\ \{\ () \to M\ \}]\!] = \mathbf{unbox}\ \mathrm{box}()\ \mathbf{as}\ \{\ () \to [\![M]\!]\ \} \mapsto ()\ \mathbf{as}\ \{\ () \to [\![M]\!]\ \} \mapsto [\![M]\!]$$

$$(\beta\times)\ \ \mathbf{match}\ (V, V')\ \mathbf{as}\ \{\ (x,y) \to M\ \} = M[V/x, V'/y]$$

$$\begin{array}{l}[\![\mathbf{match}\ (V,V')\ \mathbf{as}\ \{\ (x,y) \to M\ \}]\!]\\ = \mathbf{unbox}\ \mathrm{box}(\mathrm{val\,ref}\ [\![V]\!], \mathrm{val\,ref}\ [\![V']\!])\ \mathbf{as}\ \{\ \mathrm{val\,ref}\ x, \mathrm{val\,ref}\ y \to [\![M]\!]\ \}\\ \mapsto \mathbf{val\,ref}\ [\![V]\!], \mathrm{val\,ref}\ [\![V']\!]\ \mathbf{as}\ \{\ \mathrm{val\,ref}\ x, \mathrm{val\,ref}\ y \to [\![M]\!]\ \}\\ \mapsto [\![M]\!]\,[[\![V]\!]/x, [\![V']\!]/y] = [\![M[V/x, V'/y]]\!]\end{array}$$

$(\beta+)$ $\mathbf{match}\ (b, V)\ \mathbf{as}\ \{\ (\mathtt{0}, x_0) \to M_0; (\mathtt{1}, x_1) \to M_1\ \} = M_b[V/x_b]$

$$\begin{aligned}
&[\![\mathbf{match}\ (b, V)\ \mathbf{as}\ \{\ (\mathtt{0}, x_0) \to M_0; (\mathtt{1}, x_1) \to M_1\ \}]\!] \\
&= \mathbf{unbox}\ \mathrm{box}(b, \mathrm{val\ ref}\ [\![V]\!])\ \mathbf{as}\ \{\ \mathtt{0}, \mathrm{val\ ref}\ x_0[\![M_0]\!]; \mathtt{1}, \mathrm{val\ ref}\ x_1[\![M_1]\!]\ \} \\
&\mapsto b, \mathrm{val\ ref}\ [\![V]\!]\ \mathbf{as}\ \{\ \mathtt{0}, \mathrm{val\ ref}\ x_0[\![M_0]\!]; \mathtt{1}, \mathrm{val\ ref}\ x_1[\![M_1]\!]\ \} \\
&\mapsto [\![M_b]\!][[\![V]\!]/x_b] = [\![M_b[V/x_b]]\!]
\end{aligned}$$

$(\beta\exists)$ $\mathbf{match}\ (T, V)\ \mathbf{as}\ \{\ (X, y) \to M\ \} = M[T/X, V/y]$

$$\begin{aligned}
&[\![\mathbf{match}\ (T, V)\ \mathbf{as}\ \{\ (x, y) \to M\ \}]\!] \\
&= \mathbf{unbox}\ \mathrm{box}(\mathrm{ty}\ [\![T]\!], \mathrm{val\ ref}\ [\![V]\!])\ \mathbf{as}\ \{\ \mathrm{ty}\ X, \mathrm{val\ ref}\ y \to [\![M]\!]\ \} \\
&\mapsto \mathrm{ty}\ [\![T]\!], \mathrm{val\ ref}\ [\![V]\!]\ \mathbf{as}\ \{\ \mathrm{ty}\ X, \mathrm{val\ ref}\ y \to [\![M]\!]\ \} \\
&\mapsto [\![M]\!][[\![T]\!]/X, [\![V]\!]/y] = [\![M[T/X, V/y]]\!]
\end{aligned}$$

$(\beta\,\mathrm{F})$ $\mathbf{do}\ x \leftarrow \mathrm{return}\ V\ \mathbf{in}\ M = M[V/x]$

$$\begin{aligned}
[\![\mathbf{do}\ x \leftarrow \mathrm{return}\ V\ \mathbf{in}\ M]\!] &= \mathbf{do}\ \mathrm{ret\ val\ ref}\ [\![V]\!]\ \mathbf{as}\ \{\ \mathrm{val\ ref}\ x \to [\![M]\!]\ \} \\
&\mapsto \mathrm{val\ ref}\ [\![V]\!]\ \mathbf{as}\ \{\ \mathrm{val\ ref}\ x \to [\![M]\!]\ \} \\
&\mapsto [\![M]\!][[\![V]\!]/x] = [\![M[V/x]]\!]
\end{aligned}$$

$(\beta\to)$ $(\lambda x.M)\ V = M[V/x]$

$$\begin{aligned}
[\![(\lambda x.M)\ V]\!] &= \mathrm{proc}\ \{\ \mathrm{val\ ref}\ x \cdot \mathrm{eval\ sub} \to [\![M]\!]\ \}\ .\ \mathrm{enter}(\mathrm{val\ ref}\ [\![V]\!]).\ \mathrm{eval\ sub} \\
&\mapsto \lambda\ \{\ \mathrm{val\ ref}\ x \cdot \mathrm{eval\ sub} \to [\![M]\!]\ \}\ (\mathrm{val\ ref}\ [\![V]\!]).\ \mathrm{eval\ sub} \\
&\mapsto [\![M]\!][[\![V]\!]/x] = [\![M[V/x]]\!]
\end{aligned}$$

$(\beta\&)$ $(\lambda\ \{\ \mathtt{0}.M_0; \mathtt{1}.M_1\ \})\ b = M_b$

$$\begin{aligned}
[\![(\lambda\ \{\ \mathtt{0}.M_0; \mathtt{1}.M_1\ \})\ b]\!] &= \mathrm{proc}\ \{\ \mathtt{0} \cdot \mathrm{eval\ sub} \to [\![M_0]\!]; \mathtt{1} \cdot \mathrm{eval\ sub} \to [\![M_1]\!]\ \}\ .\ \mathrm{enter}\ b.\ \mathrm{eval\ sub} \\
&\mapsto \lambda\ \{\ \mathtt{0} \cdot \mathrm{eval\ sub} \to [\![M_0]\!]; \mathtt{1} \cdot \mathrm{eval\ sub} \to [\![M_1]\!]\ \}\ \ b.\ \mathrm{eval\ sub} \\
&\mapsto [\![M_b]\!]
\end{aligned}$$

$(\beta\forall)$ $(\Lambda X.M)\ T = M[T/X]$

$$\begin{aligned}
[\![(\Lambda X.M)\ T]\!] &= \mathrm{proc}\ \{\ \mathrm{ty}\ X \cdot \mathrm{eval\ sub} \to [\![M]\!]\ \}\ .\ \mathrm{enter}(\mathrm{ty}\ [\![T]\!]).\ \mathrm{eval\ sub} \\
&\mapsto \lambda\ \{\ \mathrm{ty}\ X \cdot \mathrm{eval\ sub} \to [\![M]\!]\ \}\ (\mathrm{ty}\ [\![T]\!]).\ \mathrm{eval\ sub} \\
&\mapsto [\![M]\!][[\![T]\!]/X] = [\![M[T/X]]\!]
\end{aligned}$$

$(\beta\,\mathrm{U})$ $(\mathrm{thunk}\ M).\ \mathrm{force} = M$

$$\begin{aligned}
[\![(\mathrm{thunk}\ M).\ \mathrm{force}]\!] &= \mathrm{clos}\ \{\ \mathrm{eval\ sub} \to [\![M]\!]\ \}\ .\ \mathrm{call}\ .\ \mathrm{eval\ sub} \\
&\mapsto \lambda\ \{\ \mathrm{eval\ sub} \to [\![M]\!]\ \}\ .\ \mathrm{eval\ sub} \\
&\mapsto [\![M]\!]
\end{aligned}$$

Soundness of η laws:

$(\eta\to)$ $\lambda x.M\ x = M : A \to \underline{B}$

$$\begin{aligned}
[\![\lambda x.M\ x]\!] &= \mathrm{proc}\ \{\ \mathrm{val\ ref}\ x \cdot \mathrm{eval\ sub} \to [\![M]\!].\ \mathrm{enter}(\mathrm{val\ ref}\ x).\ \mathrm{eval\ sub}\ \} \\
&=_{\eta\,\mathrm{Proc}} [\![M]\!]
\end{aligned}$$

$(\eta\top)$ $\lambda\ \{\ \} = M : \top$

$$[\![\lambda\ \{\ \}]\!] = \mathrm{proc}\ \{\ \} =_{\eta\,\mathrm{Proc}} [\![M]\!]$$

$(\eta\&)$ $\lambda\,\{\,\mathtt{0}.M\;\mathtt{0}; \mathtt{1}.M\;\mathtt{1}\,\} = M : \underline{A}\,\&\,\underline{B}$

$$\begin{aligned}[\![\lambda\,\{\,\mathtt{0}.M\;\mathtt{0}; \mathtt{1}.M\;\mathtt{1}\,\}]\!] &= \mathrm{proc}\,\{\mathtt{0}\cdot \mathrm{eval\,sub} \to [\![M]\!].\,\mathrm{enter}\,\mathtt{0}.\,\mathrm{eval\,sub};\\ &\qquad\qquad \mathtt{1}\cdot \mathrm{eval\,sub} \to [\![M]\!].\,\mathrm{enter}\,\mathtt{1}.\,\mathrm{eval\,sub}\}\\ &=_{\eta\,\mathrm{Proc}} [\![M]\!]\end{aligned}$$

$(\eta\forall)$ $\Lambda X.M\;X = M : \forall X{:}\tau.\underline{A}$

$$\begin{aligned}[\![\Lambda X.M\;X]\!] &= \mathrm{proc}\,\{\mathrm{ty}\,X\cdot \mathrm{eval\,sub} \to [\![M]\!].\,\mathrm{enter}(\mathrm{ty}\,X).\,\mathrm{eval\,sub}\}\\ &=_{\eta\,\mathrm{Proc}} [\![M]\!]\end{aligned}$$

$(\eta\,\mathrm{U})$ $\mathrm{thunk}(V.\,\mathrm{force}) = V : \mathrm{U}\,\underline{A}$

$$\begin{aligned}[\![\mathrm{thunk}(V.\,\mathrm{force})]\!] &= \mathrm{clos}\,\{\,\mathrm{eval\,sub} \to [\![V]\!].\,\mathrm{call}.\,\mathrm{eval\,sub}\,\}\\ &=_{\eta\,\mathrm{Proc}} [\![V]\!]\end{aligned}$$

$(\eta 1)$ $\mathbf{match}\,V\,\mathbf{as}\,\{\,()\to M[()/z]\,\} = M[V/z]$ (if $V : 1$)

$$\begin{aligned}&[\![\mathbf{match}\,V\,\mathbf{as}\,\{\,()\to M[()/z]\,\}]\!]\\ &= \mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\,\{\,()\to [\![M[()/z]]\!]\,\}\\ &= \mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\,\{\,()\to [\![M]\!][\mathrm{box}()/z]\,\}\\ &=_{\eta\,\mathrm{Box}} [\![M]\!][[\![V]\!]/z] = [\![M[V/z]]\!]\end{aligned}$$

$(\eta\times)$ $\mathbf{match}\,V\,\mathbf{as}\,\{\,(x,y)\to M[(x,y)/z]\,\} = M[V/z]$ (if $V : A\times B$)

$$\begin{aligned}&[\![\mathbf{match}\,V\,\mathbf{as}\,\{\,(x,y)\to M[(x,y)/z]\,\}]\!]\\ &= \mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\,\{\,(\mathrm{val\,ref}\,x, \mathrm{val\,ref}\,y)\to [\![M[(x,y)/z]]\!]\,\}\\ &= \mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\,\{\,(\mathrm{val\,ref}\,x, \mathrm{val\,ref}\,y)\to [\![M]\!][\mathrm{box}(\mathrm{val\,ref}\,x, \mathrm{val\,ref}\,y)/z]\,\}\\ &=_{\eta\,\mathrm{Box}} [\![M]\!][[\![V]\!]/z] = [\![M[V/z]]\!]\end{aligned}$$

$(\eta+)$ $\mathbf{match}\,V\,\mathbf{as}\,\{\,(\mathtt{0},x)\to M[(\mathtt{0},x)/z]; (1,x)\to M[(1,x)/z]\,\} = M[V/z]$ (if $V : A+B$)

$$\begin{aligned}&\left[\!\!\left[\begin{array}{l}\mathbf{match}\,V\,\mathbf{as}\{(\mathtt{0},x)\to M[(\mathtt{0},x)/z];\\ \qquad\qquad (1,y)\to M[(1,y)/z]\}\end{array}\right]\!\!\right]\\ &= \begin{array}{l}\mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\{(\mathtt{0},\mathrm{val\,ref}\,x)\to [\![M[(\mathtt{0},x)/z]]\!];\\ \qquad\qquad (1,\mathrm{val\,ref}\,y)\to [\![M[(1,y)/z]]\!]\}\end{array}\\ &= \begin{array}{l}\mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\,\{(\mathtt{0},\mathrm{val\,ref}\,x)\to [\![M]\!][(\mathtt{0},\mathrm{val\,ref}\,x)/z];\\ \qquad\qquad (1,\mathrm{val\,ref}\,y)\to [\![M]\!][(1,\mathrm{val\,ref}\,y)/z]\}\end{array}\\ &=_{\eta\,\mathrm{Box}} [\![M]\!][[\![V]\!]/z] = [\![M[V/z]]\!]\end{aligned}$$

$(\eta 0)$ $\mathbf{match}\,V\,\mathbf{as}\,\{\,\} = M$ (if $V : 0$)

$$[\![\mathbf{match}\,V\,\mathbf{as}\,\{\,\}]\!] = \mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\,\{\,\} =_{\eta\,\mathrm{Box}} [\![M]\!]$$

$(\eta\exists)$ $\mathbf{match}\,V\,\mathbf{as}\,\{\,(X,y)\to M[(X,y)/z]\,\} = M[V/z]$ (if $V : \exists X{:}\tau.A$)

$$\begin{aligned}&[\![\mathbf{match}\,V\,\mathbf{as}\,\{\,(X,y)\to M[(X,y)/z]\,\}]\!]\\ &= \mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\,\{\,(\mathrm{ty}\,X, \mathrm{val\,ref}\,y)\to [\![M[(X,y)/z]]\!]\,\}\\ &= \mathbf{unbox}\,[\![V]\!]\,\mathbf{as}\,\{\,(\mathrm{ty}\,X, \mathrm{val\,ref}\,y)\to [\![M]\!][\mathrm{box}(\mathrm{ty}\,X, \mathrm{val\,ref}\,y)/z]\,\}\\ &=_{\eta\,\mathrm{Box}} [\![M]\!][[\![V]\!]/z] = [\![M[V/z]]\!]\end{aligned}$$

$(\eta\,\mathrm{F})$ $\mathbf{do}\,x \leftarrow M\,\mathbf{in}\,\mathrm{return}\,x = M$ (if $M : \mathrm{F}\,A$)

$$\begin{aligned}
&[\![\mathbf{do}\,x \leftarrow M\,\mathbf{in}\,\mathrm{return}\,x = M]\!] \\
&= \mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,x \to \mathrm{ret\,val\,ref}\,x\,\} \\
&=_{\eta\,\mathrm{Ret}_{Id}} [\![M]\!]
\end{aligned}$$

Sequencing laws:

$(cc\,\mathrm{F})$ $\mathbf{do}\,y \leftarrow (\mathbf{do}\,x \leftarrow M\,\mathbf{in}\,M')\,\mathbf{in}\,M'' = \mathbf{do}\,x \leftarrow M\,\mathbf{in}(\mathbf{do}\,y \leftarrow M'\,\mathbf{in}\,M'')$

$$\begin{aligned}
&[\![\mathbf{do}\,y \leftarrow (\mathbf{do}\,x \leftarrow M\,\mathbf{in}\,M')\,\mathbf{in}\,M'']\!] \\
&= \mathbf{do}\,(\mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,x \to [\![M']\!]\,\})\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,y \to [\![M'']\!]\,\} \\
&=_{cc\,\mathrm{Ret}} \mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,x \to \mathbf{do}\,[\![M']\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,y \to [\![M'']\!]\,\}\,\} \\
&= [\![\mathbf{do}\,x \leftarrow M\,\mathbf{in}(\mathbf{do}\,y \leftarrow M'\,\mathbf{in}\,M'')]\!]
\end{aligned}$$

$(cc{\to})$ $\mathbf{do}\,y \leftarrow M\,\mathbf{in}(\lambda x.M') = \lambda x.(\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,M')$

$$\begin{aligned}
&[\![\mathbf{do}\,y \leftarrow M\,\mathbf{in}(\lambda x.M')]\!] \\
&= \mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,y \to \mathrm{proc}\,\{\,\mathrm{val\,ref}\,x \cdot \mathrm{eval\,sub} \to [\![M']\!]\,\}\,\} \\
&=_{cc\,\mathrm{Proc}} \mathrm{proc}\,\{\,\mathrm{val\,ref}\,x \cdot \mathrm{eval\,sub} \to \mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,y \to [\![M']\!]\,\}\,\} \\
&= [\![\lambda x.(\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,M')]\!]
\end{aligned}$$

$(cc\&)$ $\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,\lambda\,\{\,\mathtt{0}.M_0; \mathtt{1}.M_1\,\} = \lambda\,\{\,\mathtt{0}.\,\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,M_0; \mathtt{1}.\,\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,M_1\,\}$

$$\begin{aligned}
&[\![\mathbf{do}\,y \leftarrow M\,\mathbf{in}(\lambda\,\{\,\mathtt{0}.M_0; \mathtt{1}.M_1\,\})]\!] \\
&= \mathsf{do}\,[\![M]\!]\,\mathsf{as}\,\{\mathrm{val\,ref}\,y \to \mathrm{proc}\,\{\mathtt{0} \cdot \mathrm{eval\,sub} \to [\![M_0]\!]; \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{1} \cdot \mathrm{eval\,sub} \to [\![M_1]\!]\}\} \\
&=_{cc\,\mathrm{Proc}} \mathrm{proc}\,\{\mathtt{0} \cdot \mathrm{eval\,sub} \to \mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,y \to [\![M_0]\!]\,\}; \\
&\qquad\qquad\qquad\qquad \mathtt{1} \cdot \mathrm{eval\,sub} \to \mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,y \to [\![M_1]\!]\,\}\} \\
&= [\![\lambda\,\{\,\mathtt{0}.\,\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,M_0; \mathtt{1}.\,\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,M_1\,\}]\!]
\end{aligned}$$

$(cc\forall)$ $\mathbf{do}\,y \leftarrow M\,\mathbf{in}(\Lambda X.M') = \Lambda X.(\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,M')$

$$\begin{aligned}
&[\![\mathbf{do}\,y \leftarrow M\,\mathbf{in}(\Lambda X.M')]\!] \\
&= \mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,y \to \mathrm{proc}\,\{\,\mathrm{ty}\,X \cdot \mathrm{eval\,sub} \to [\![M']\!]\,\}\,\} \\
&=_{cc\,\mathrm{Proc}} \mathrm{proc}\,\{\,\mathrm{ty}\,X \cdot \mathrm{eval\,sub} \to \mathbf{do}\,[\![M]\!]\,\mathbf{as}\,\{\,\mathrm{val\,ref}\,y \to [\![M']\!]\,\}\,\} \\
&= [\![\Lambda X.(\mathbf{do}\,y \leftarrow M\,\mathbf{in}\,M')]\!]
\end{aligned}$$

□

To show completeness, we need to be able to translate back from the image of $CBPV[\![\_]\!]^{-1}$: the uniform subsyntax shown in fig. 19. To do so, we only need to delete all the boxes (Box and Proc) and representation information, and treat the two forms $\mathrm{proc}\,F$ and λF as equivalent. The inverse translation is given in fig. 20.

Lemma D.3 (Inverse Type Soundness). *Let $[\![\_]\!]^{-1}$ denote $CBPV[\![\_]\!]^{-1}$ in the following:*
(1) If $\Gamma \vdash T : \tau$ in Uniform CBUV then $[\![\Gamma]\!]^{-1} \vdash [\![T]\!]^{-1} : [\![\tau]\!]^{-}$ in Polymorphic CBUV.
(2) If $\Gamma \vdash V : A : \mathrm{ref}\,\mathbf{val}$ in Uniform CBUV then $[\![\Gamma]\!]^{-1} \vdash [\![V]\!]^{-1} : [\![A]\!]^{-1}$ in Polymorphic CBUV.
(3) If $\Gamma \vdash S : A$ in Uniform CBUV then $[\![\Gamma]\!]^{-1} \vdash [\![S]\!]^{-1} : [\![A]\!]^{-1}$ in Polymorphic CBUV.
(4) If $\Gamma \vdash M : B : \mathrm{sub}\,\mathbf{comp}$ in Uniform CBUV then $[\![\Gamma]\!]^{-1} \vdash [\![M]\!]^{-1} : [\![B]\!]^{-1}$ in Polymorphic CBPV.
(5) If $\Gamma \vdash L : B$ in Uniform CBUV then $[\![\Gamma]\!]^{-1} \vdash [\![L]\!]^{-1} : [\![B]\!]^{-1}$ in Polymorphic CBPV.

Proof (sketch). By (mutual) induction on the given typing derivations. □

Syntax of complex values and complex computations:

$$
\begin{aligned}
\mathit{StructShape} \ni s &::= () \mid \mathrm{val}\,\square, \mathrm{val}\,\square \mid b, \mathrm{val}\,\square \mid \square, \mathrm{val}\,\square \\
\mathit{Struct} \ni S &::= s[\delta] \\
\mathit{Pattern} \ni p &::= s[\gamma] \\
\mathit{MatchCode} \ni G &::= \{\, p \to M \ldots \,\} \\
\mathit{Bit} \ni b &::= \mathtt{0} \mid \mathtt{1} \\
\mathit{StackShape} \ni k &::= \mathrm{val}\,\square \cdot \mathrm{eval}\,\mathrm{sub} \mid b \cdot \mathrm{eval}\,\mathrm{sub} \mid \square \cdot \mathrm{eval}\,\mathrm{sub} \\
\mathit{Stack} \ni K &::= k[\delta] \\
\mathit{Copattern} \ni q &::= k[\gamma] \\
\mathit{FunCode} \ni F &::= \{\, q \to M \ldots \,\} \\
\mathit{Call} \ni L &::= \lambda F \mid M.\,\mathrm{enter}
\end{aligned}
$$

Syntax of atomic values and computations:

$$
\begin{aligned}
\mathit{Value} \ni V &::= \mathrm{ref}\,x \mid \mathrm{box}\,S \mid \mathrm{clos}\,\{\,\mathrm{eval}\,\mathrm{sub} \to M\,\} \mid T \\
\mathit{Args} \ni \delta &::= \bullet \mid \delta, V \mid \delta, T \\
\mathit{Params} \ni \gamma &::= \bullet \mid \gamma, \mathrm{ref}\,x : A \mid \gamma, \mathrm{ty}\,X : \mathrm{Type}\,\tau \\
\mathit{Comp} \ni M &::= S\ \mathbf{as}\ G \mid \mathbf{unbox}\ V\ \mathbf{as}\ G \mid \mathbf{do}\ M\ \mathbf{as}\ \{\,\mathrm{val}\,\mathrm{ref}\,x \to M\,\} \mid \mathrm{val}\,V\ \mathbf{as}\ \{\,\mathrm{val}\,\mathrm{ref}\,x \to M\,\} \\
&\quad \mid \mathrm{ret}\,\mathrm{val}\,V \mid \mathrm{proc}\,F \mid \langle L \parallel K \rangle \mid \langle V.\,\mathrm{call} \parallel \mathrm{eval}\,\mathrm{sub} \rangle \mid \langle \lambda\,\{\,\mathrm{eval}\,\mathrm{sub} \to M\,\} \parallel \mathrm{eval}\,\mathrm{sub} \rangle
\end{aligned}
$$

Syntax of types:

$$
\begin{aligned}
\mathit{Type} \ni T &::= A \mid B \mid P \mid Q \\
\mathit{Kind} \ni \tau &::= \mathrm{ref}\ \mathbf{val} \mid \mathbf{cplx\ val} \mid \mathrm{sub}\ \mathbf{comp} \mid \mathbf{cplx\ comp} \\
\mathit{CmplxValTy} \ni P &::= x \mid 1 \mid \mathrm{Val}\,A_0 \times \mathrm{Val}\,A_1 \mid 0 \mid \mathrm{Val}\,A_0 + \mathrm{Val}\,A_1 \mid \exists R\ x : A.\ \mathrm{Val}\,A \\
\mathit{AtomValTy} \ni A &::= x \mid \mathrm{Box}\,P \mid \mathrm{Clos}(\mathrm{Eval}\,B) \mid \mathrm{Type}\,\tau \\
\mathit{CmplxCompTy} \ni Q &::= x \mid \mathrm{Val}\,A \to \mathrm{Eval}\,B \mid \top \mid \mathrm{Eval}\,B_0\ \&\ \mathrm{Eval}\,B_1 \mid \forall R\ x : A.\ \mathrm{Eval}\,B \\
\mathit{AtomCompTy} \ni B &::= x \mid \mathrm{Ret}(\mathrm{Val}\,A) \mid \mathrm{Proc}\,Q
\end{aligned}
$$

Fig. 19. The Uniform subsyntax of Call-By-Unboxed-Value.

Lemma D.4 (Inverse Equational Soundness). *Equality within the uniform subsyntax Call-By-Unboxed-Value in is sound with respect to Polymorphic Call-By-Push-Value: for any M and M' defined in fig. 19, if $M =_{\beta\eta} M'$ then $CBPV[\![M]\!]^{-1} =_{\beta\eta cc} CBPV[\![M']\!]^{-1}$, and similarly for values, types,* etc.*in fig. 19.*

Proof. Let $[\![\_]\!]^{-1}$ denote $CBPV[\![\_]\!]^{-1}$ in the following.
Soundness of β reduction rules.

- β Box, β Clos, β Proc, and β Ret: both sides translate to identical terms.
- β **as**: depends on the structure of S.
  - $S = \mathrm{val}\,V$: $\mathrm{val}\,V\ \mathbf{as}\ \{\,\mathrm{val}\,\mathrm{ref}\,x \to M\,\} \mapsto M[V/x]$

$$
\begin{aligned}
[\![\mathbf{do}\ \mathrm{ret}\,\mathrm{val}\,V\ \mathbf{as}\ \{\,\mathrm{val}\,\mathrm{ref}\,x \to M\,\}]\!]^{-1} &= \mathbf{do}\ x \leftarrow \mathrm{return}\ [\![V]\!]^{-1}\ \mathbf{in}\ [\![M]\!]^{-1} \\
&\mapsto_{\beta\,\mathrm{Ret}} [\![M]\!]^{-1}[V/x] \\
&= [\![M[V/x]]\!]^{-1}
\end{aligned}
$$

Translation of structure and stack shapes:

$$CBPV[\![()]\!]^{-1} = ()$$

$$CBPV[\![(\mathrm{val}\,\square, \mathrm{val}\,\square)]\!]^{-1} = (\square, \square) \qquad CBPV[\![\mathrm{val}\,\square \cdot \mathrm{eval\,sub}]\!]^{-1} = \square\ \square$$

$$CBPV[\![(b, \mathrm{val}\,\square)]\!]^{-1} = (b, \square) \qquad CBPV[\![b \cdot \mathrm{eval\,sub}]\!]^{-1} = \square\ b$$

$$CBPV[\![(\square, \mathrm{val}\,\square)]\!]^{-1} = (\square, \square) \qquad CBPV[\![\square \cdot \mathrm{eval\,sub}]\!]^{-1} = \square\ \square$$

Translation of filled structures and stacks, (co)patterns, and (co)pattern-matching code:

$$CBPV[\![s[\delta]]\!]^{-1} = CBPV[\![s]\!]^{-1}[CBPV[\![\delta]\!]^{1}] \qquad CBPV[\![s[\gamma]]\!]^{-1} = CBPV[\![s]\!]^{-1}[CBPV[\![\gamma]\!]^{-1}]$$

$$CBPV[\![k[\delta]]\!]^{-1} = CBPV[\![k]\!]^{-1}[CBPV[\![\delta]\!]^{1}] \qquad CBPV[\![k[\gamma]]\!]^{-1} = CBPV[\![k]\!]^{-1}[CBPV[\![\gamma]\!]^{-1}]$$

$$CBPV[\![\{\ p \to M \ldots\ \}]\!]^{-1} = \{\ CBPV[\![p]\!]^{-1} \to CBPV[\![M]\!]^{-1} \ldots\ \}$$

$$CBPV[\![\{\ q \to M \ldots\ \}]\!]^{-1} = \{\ CBPV[\![q]\!]^{-1}.\ CBPV[\![M]\!]^{-1} \ldots\ \}$$

$$CBPV[\![\delta, V]\!]^{-1} = CBPV[\![\delta]\!]^{-1}, CBPV[\![V]\!]^{-1} \qquad CBPV[\![\gamma, \mathrm{ref}\ x : A]\!]^{-1} = CBPV[\![\gamma]\!]^{-1}, x : CBPV[\![A]\!]^{-1}$$

$$CBPV[\![\delta, T]\!]^{-1} = CBPV[\![\delta]\!]^{-1}, CBPV[\![T]\!]^{-1} \qquad CBPV[\![\gamma, \mathrm{ty}\ X : \tau]\!]^{-1} = CBPV[\![\gamma]\!]^{-1}, X : CBPV[\![\tau]\!]^{-1}$$

$$CBPV[\![\bullet]\!]^{-1} = \bullet$$

Translation of atomic values:

$$CBPV[\![\mathrm{ref}\ x]\!]^{-1} = x \qquad CBPV[\![\mathrm{box}\ S]\!]^{-1} = CBPV[\![S]\!]^{-1}$$

$$CBPV[\![T]\!]^{-1} = CBPV[\![T]\!]^{-1} \qquad CBPV[\![\mathrm{clos}\ \{\ \mathrm{eval\,sub} \to M\ \}]\!]^{-1} = \mathrm{thunk}\ CBPV[\![M]\!]^{-1}$$

Translation of complex calls:

$$CBPV[\![\lambda\ \{\ \mathrm{val\,ref}\ x : A \cdot \mathrm{eval\,sub} \to M\ \}]\!]^{-1} = \lambda x : CBPV[\![A]\!]^{-1}.CBPV[\![M]\!]^{-1}$$

$$CBPV[\![\lambda\ \{\ b \cdot \mathrm{eval\,sub} \to M_b{}^{b \in \{0,1\}}_{\ldots}\ \}]\!]^{-1} = \lambda\ \{\ b.\ CBPV[\![M_b]\!]^{-1}{}^{b \in \{0,1\}}_{\ldots}\ \}$$

$$CBPV[\![\lambda\ \{\ \}]\!]^{1} = \lambda\ \{\ \}$$

$$CBPV[\![\lambda\ \{\ \mathrm{ty}\ X : \tau \to M\ \}]\!]^{-1} = \Lambda X : CBPV[\![\tau]\!]^{-1}.CBPV[\![M]\!]^{-1}$$

$$CBPV[\![M.\ \mathrm{enter}]\!]^{-1} = CBPV[\![M]\!]^{-1}$$

Translation of atomic computations:

$$CBPV[\![\mathbf{unbox}\ V\ \mathbf{as}\ G]\!]^{-1} = \mathbf{match}\ CBPV[\![V]\!]^{-1}\ \mathbf{as}\ CBPV[\![G]\!]^{-1}$$

$$CBPV[\![S\ \mathbf{as}\ G]\!]^{-1} = \mathbf{match}\ CBPV[\![S]\!]^{-1}\ \mathbf{as}\ CBPV[\![G]\!]^{-1}$$

$$CBPV[\![\mathbf{do}\ M\ \mathbf{as}\ \{\ \mathrm{val\,ref}\ x : A \to M'\ \}]\!]^{-1} = \mathbf{do}\ x{:}CBPV[\![A]\!]^{-1} \leftarrow CBPV[\![M]\!]^{-1}\ \mathbf{in}\ CBPV[\![M']\!]^{-1}$$

$$CBPV[\![\mathrm{val}\ V\ \mathbf{as}\ \{\ \mathrm{val\,ref}\ x : A \to M\ \}]\!]^{-1} = \mathbf{do}\ x{:}CBPV[\![A]\!]^{-1} \leftarrow \mathrm{return}\ CBPV[\![V]\!]^{-1}\ \mathbf{in}\ CBPV[\![M]\!]^{-1}$$

$$CBPV[\![\mathrm{ret\,val}\ V]\!]^{-1} = \mathrm{return}\ CBPV[\![V]\!]^{-1}$$

$$CBPV[\![\mathrm{proc}\ F]\!]^{-1} = CBPV[\![\lambda F]\!]^{-1}$$

$$CBPV[\![\langle L \parallel K\rangle]\!]^{-1} = CBPV[\![K]\!]^{-1}[CBPV[\![L]\!]^{-1}]$$

$$CBPV[\![\langle V.\ \mathrm{call} \parallel \mathrm{eval\,sub}\rangle]\!]^{-1} = CBPV[\![V]\!]^{-1}.\ \mathrm{force}$$

$$CBPV[\![\langle \lambda\ \{\ \mathrm{eval\,sub} \to M\ \} \parallel \mathrm{eval\,sub}\rangle]\!]^{-1} = (\mathrm{thunk}\ CBPV[\![M]\!]^{-1}).\ \mathrm{force}$$

Fig. 20. Inverse translation from Uniform Call-By-Unboxed-Value to Polymorphic Call-By-Push-Value.

– In all other cases, reduction follows by the type-specific rule matching S. For example,

$$[\![(\mathtt{0}, V)\ \mathbf{as}\ \{\ (b, \mathrm{ref}\ x_b) \to M_b{}^{b \in \{0,1\}}_{\ldots}\ \}]\!]^{-1} = \mathbf{match}(\mathtt{0}, [\![V]\!]^{-1})\ \mathbf{as}\ \{\ (b, x_b) \to [\![M_b]\!]^{-1}{}^{b \in \{0,1\}}_{\ldots}\ \}$$
$$\mapsto_{\beta+} [\![M_0]\!]^{-1}$$

- $\beta\lambda$: depends on the structure of K.
  - $K = \text{eval sub}$: $\langle \lambda \{ \text{eval sub} \to M \} \parallel \text{eval sub} \rangle \mapsto M$

$$[\![\langle \lambda \{ \text{eval sub} \to M \} \parallel \text{eval sub} \rangle]\!]^{-1} = (\text{thunk } [\![M]\!]^{-1}).\,\text{force} \mapsto_{\beta \mathrm{U}} [\![M]\!]^{-1}$$

  - In all other cases, reduction follows by the type-specific rule matching K. For example,

$$\begin{aligned}
&[\![\langle \lambda \{ \text{val ref } x \cdot \text{eval sub} \to M \} \parallel \text{val ref } V \cdot \text{eval sub} \rangle]\!]^{-1} \\
&= (\lambda x.[\![M]\!]^{-1})\ [\![V]\!]^{-1} \\
&\mapsto_{\beta\to} [\![M]\!]^{-1}[[\![V]\!]^{-1}/x] \\
&= [\![M[V/x]]\!]^{-1}
\end{aligned}$$

Soundness of η axioms.

- η ret: Note that $E[\mathbf{do}\, x \leftarrow M \,\mathbf{in}\, M'] = \mathbf{do}\, x \leftarrow M \,\mathbf{in}\, E[M']$ is derivable from the β, η, and cc axioms in Call-By-Push-Value by induction on E:
  - $E = \square$: trivial
  - $E = \mathbf{do}\, y \leftarrow E' \,\mathbf{in}\, M''$:

$$\begin{aligned}
&\mathbf{do}\, y \leftarrow E'[\mathbf{do}\, x \leftarrow M \,\mathbf{in}\, M'] \,\mathbf{in}\, M'' \\
&=_{IH} \mathbf{do}\, y \leftarrow (\mathbf{do}\, x \leftarrow M \,\mathbf{in}\, E'[M']) \,\mathbf{in}\, M'' \\
&=_{cc\,\mathrm{F}} \mathbf{do}\, x \leftarrow M \,\mathbf{in}\, \mathbf{do}\, y \leftarrow E'[M'] \,\mathbf{in}\, M''
\end{aligned}$$

  - $E = E'\ V$:

$$\begin{aligned}
&E'[\mathbf{do}\, x \leftarrow M \,\mathbf{in}\, M']\ V \\
&=_{IH} \mathbf{do}\, x \leftarrow M \,\mathbf{in}\, E'[M']\ V \\
&=_{\eta\to} (\mathbf{do}\, x \leftarrow M \,\mathbf{in}\, \lambda x.(E'[M']\ x))\ V \\
&=_{cc\to} (\lambda x.(\mathbf{do}\, x \leftarrow M \,\mathbf{in}\, E'[M']\ x))\ V \\
&=_{\beta\to} \mathbf{do}\, x \leftarrow M \,\mathbf{in}\, (E'[M']\ V)
\end{aligned}$$

  - Other cases ($E'\ \mathtt{0}$, $E'\ \mathtt{1}$, $E'\ T$) are similar to $E'\ V$

  $\mathbf{do}\, M \,\mathbf{as}\, \{ \text{val ref } x \to E[\text{ret val ref } x] \} = E[M]$ follows from the above equation, since $[\![E[M]]\!]^{-1}$ decomposes into $[\![E]\!]^{-1}[[\![M]\!]^{-1}]$ where $[\![E]\!]^{-1}$ is a Call-By-Push-Value evaluation context:

$$\begin{aligned}
&[\![\mathbf{do}\, M \,\mathbf{as}\, \{ \text{val ref } x \to E[\text{ret val ref } x] \}]\!]^{-1} \\
&= \mathbf{do}\, x \leftarrow [\![M]\!]^{-1} \,\mathbf{in}\, [\![E]\!]^{-1}[\text{return } x] \\
&=_{\beta\eta cc} [\![E]\!]^{-1}[\mathbf{do}\, x \leftarrow [\![M]\!]^{-1} \,\mathbf{in}\, \text{return } x] \\
&=_{\eta\,\mathrm{F}} [\![E]\!]^{-1}[[\![M]\!]^{-1}] = [\![E[M]]\!]^{-1}
\end{aligned}$$

- η Clos: $\text{clos} \{ \text{eval sub} \to \langle V.\,\text{call} \parallel \text{eval sub} \rangle \} = V$

$$[\![\text{clos} \{ \text{eval sub} \to \langle V.\,\text{call} \parallel \text{eval sub} \rangle \}]\!]^{-1} = \text{thunk}([\![V]\!]^{-1}.\,\text{force}) =_{\eta\,\mathrm{U}} [\![V]\!]^{-1}$$

- η Proc: each special case in the image of $CBPV[\![\_]\!]$ follows from one of the axioms: $\eta\to$, $\eta\&$, $\eta\top$, or $\eta\forall$.
- η Box: each special case in the image of $CBPV[\![\_]\!]$ follows from one of the axioms: $\eta\times$, $\eta 1$, $\eta +$, $\eta 0$, $\eta\exists$. □

Lemma D.5 (Round-Trip Identity). *(1) All Polymorphic Call-By-Push-Value computations, values, and types are α-equivalent to their round-trip translation: $CBPV[\![CBPV[\![M]\!]]\!]^{-1} =_\alpha M$, and $CBPV[\![CBPV[\![V]\!]]\!]^{-1} =_\alpha V$, and $CBPV[\![CBPV[\![T]\!]]\!]^{-1} =_\alpha T$.*
(2) All atomic values, computations, and types in the uniform subsyntax of Call-By-Unboxed-Value β-equivalent to their round-trip translation: $CBPV[\![CBPV[\![M]\!]^{-1}]\!] =_\beta M$, etc.

Proof (sketch). By induction on the syntax. In particular, the second round trip, starting from the uniform subsyntax will convert S **as** G to **unbox** box S **as** G, and convert $\langle \lambda F \parallel K \rangle$ to $\langle \text{proc } F.\, \text{enter} \parallel K \rangle$, which are β-equivalent to their original forms. □

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

Proof. Follows from lemmas D.1 and D.3, since every round-trip starting from Call-By-Push-Value is the same expression (lemma D.5). □

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. $CBPV[\![M]\!] = CBPV[\![M']\!]$.*

Proof. The soundness direction is exactly lemma D.2. For completeness, assuming $CBPV[\![M]\!] =_{\beta\eta} CBPV[\![M']\!]$ we have from lemmas D.4 and D.5:

$$M =_\alpha CBPV[\![CBPV[\![M]\!]]\!]^{-1} =_{\beta\eta cc} CBPV[\![CBPV[\![M']\!]]\!]^{-1} =_\alpha M'$$ □

E PROOF OF MACHINE BISIMULATION

Because of the definition of the bisimulation relationship as $M \sim m$ if and only if $M = AM[\![m]\!]^{-1}$, the backward simulation direction is straightforward.

Lemma E.1. *If $W\ldots; \sigma' = \rho^*(V\ldots)$, then $([\![W]\!]^{-1}\ldots)[AM[\![\sigma']\!]^{-1}] = ([\![V]\!]^{-1}\ldots)[AM[\![\rho]\!]^{-1}]$.*

Proof (sketch). By induction on the sequence $V\ldots$ and definition of $\rho^*(V\ldots)$. □

Lemma E.2. *If $H; \sigma' = \sigma_\rho(V)$, then $AM[\![H]\!]^{-1}[AM[\![\sigma, \sigma']\!]^{-1}] = AM[\![V]\!]^{-1}[AM[\![\rho]\!]^{-1}, AM[\![\sigma]\!]^{-1}]$.*

Proof (sketch). By cases on the value V and definition of $\sigma_\rho(V)$. □

Lemma E.3 (Backward simulation). *(1) If $M \sim m$ and $m \mapsto_{\mathbf{do},\text{enter},\mathbf{as},\lambda} m'$ then $M \sim m'$.*
(2) If $M \sim m$ and $m \mapsto m'$ by any other rule (besides **do**, enter, **as**, *or λ), then there is an $M' \sim m'$ such that $M \mapsto M'$.*

Proof. By cases on the reduction rule $m \mapsto m'$, where we let $[\![\_]\!]^{-1}$ denote $AM[\![\_]\!]^{-1}$. In the cases for part (1), $M \sim m$ and $M \sim m'$ comes from $[\![m]\!]^{-1} = [\![m']\!]^{-1}$

(**do**)

$$\begin{aligned}
&[\![\mathbf{do}\ M\ \mathbf{as}\ G[\Gamma, O][\rho\kappa][\sigma]]\!]^{-1} \\
&= ([\![\kappa]\!]^{-1}_\sigma \left[(\mathbf{do}\ [\![M]\!]^{-1}\ \mathbf{as}\ [\![G]\!]^{-1})[[\![\rho]\!]^{-1}]\right])\ [[\![\sigma]\!]^{-1}] \\
&= [\![\langle M \rangle\ [\rho\ \text{sub}\ \overline{x}][\overline{x} := \mathbf{do}\ G[\rho|_\Gamma \kappa(O)], \sigma]]\!]^{-1}
\end{aligned}$$

(enter) where $W\ldots;\sigma' = \rho^*(V\ldots)$, by lemma E.1

$$\begin{aligned}
&[\![\langle\langle M.\,\mathrm{enter}\,\|k[V\ldots,O]\rangle\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\,\left[\langle [\![M]\!]^{-1}.\,\mathrm{enter}\,\|k[[\![V]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma]\!]^{-1}]\\
&= \left([\![\kappa]\!]_{\sigma,\sigma'}^{-1}\,\left[\langle [\![M]\!]^{-1}.\,\mathrm{enter}\,\|k[[\![W]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma,\sigma']\!]^{-1}]\\
&= [\![\langle M\rangle\,[\rho\,\mathrm{sub}\,\overline{x}]\,[\overline{x} := \mathrm{enter}\,k[W\ldots\kappa(O)],\sigma,\sigma']]\!]^{-1}
\end{aligned}$$

(**as**) where $W\ldots;\sigma' = \rho^*(V\ldots)$, by lemma E.1

$$\begin{aligned}
&[\![\langle s[V\ldots]\,\mathbf{as}\,G\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\,\left[(s[[\![V]\!]^{-1}\ldots]\,\mathbf{as}\,[\![G]\!]^{-1})[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma]\!]^{-1}]\\
&= \left([\![\kappa]\!]_{\sigma,\sigma'}^{-1}\,\left[(s[[\![W]\!]^{-1}\ldots]\,\mathbf{as}\,[\![G]\!]^{-1})[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma,\sigma']\!]^{-1}]\\
&= [\![\langle s\,\|\,W\ldots\,\mathbf{as}\,G\rangle\,[\rho\kappa]\,[\sigma,\sigma']]\!]^{-1}
\end{aligned}$$

(λ) where $W\ldots;\sigma' = \rho^*(V\ldots)$, by lemma E.1

$$\begin{aligned}
&[\![\langle\langle\lambda F\,\|\,k[V\ldots]\rangle\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\,\left[\langle\lambda[\![F]\!]^{-1}\,\|\,k[[\![V]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma]\!]^{-1}]\\
&= \left([\![\kappa]\!]_{\sigma,\sigma'}^{-1}\,\left[\langle\lambda[\![F]\!]^{-1}\,\|\,k[[\![W]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma,\sigma']\!]^{-1}]\\
&= [\![\langle F\,\|\,k\,\|\,W\ldots\rangle\,[\rho\kappa]\,[\sigma,\sigma']]\!]^{-1}
\end{aligned}$$

For the remaining cases, we have to follow one reduction step in the source, like so:

(Box) where $\mathrm{box}\,s[W\ldots];\sigma' = \sigma_\rho(V)$, by lemma E.2

$$\begin{aligned}
&[\![\langle\mathbf{unbox}\,V\,\mathbf{as}\,G\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\,\left[(\mathbf{unbox}\,[\![V]\!]^{-1}\,\mathbf{as}\,[\![G]\!]^{-1})[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma]\!]^{-1}]\\
&= \left([\![\kappa]\!]_\sigma^{-1}\,\left[(\mathbf{unbox}\,\mathrm{box}\,s[[\![W]\!]^{-1}]\,\mathbf{as}\,[\![G]\!]^{-1})[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma,\sigma']\!]^{-1}]\\
&\mapsto_{\beta\,\mathrm{Box}} \left([\![\kappa]\!]_\sigma^{-1}\,\left[(s[[\![W]\!]^{-1}]\,\mathbf{as}\,[\![G]\!]^{-1})[[\![\rho]\!]^{-1}]\right]\right)\,[[\![\sigma,\sigma']\!]^{-1}]\\
&= [\![\langle s\,\|\,W\ldots\,\|\,G\rangle\,[\rho\kappa]\,[\sigma\sigma']]\!]^{-1}
\end{aligned}$$

(Ret) where $W\ldots;\sigma' = \rho^*(V\ldots)$, by lemma E.1

$$\begin{aligned}
&[\![\langle\mathrm{ret}\,s[V\ldots]\rangle\,[\rho\,\mathrm{sub}\,\overline{x}]\,[\overline{x} := \mathbf{do}\,G[\rho'\kappa'],\sigma]]\!]^{-1}\\
&= \left([\![\kappa']\!]_\sigma^{-1}\,\left[\mathbf{do}\,\mathrm{ret}\,s[[\![V]\!]^{-1}\ldots][[\![\rho]\!]^{-1}]\,\mathbf{as}\,[\![G]\!]^{-1}[[\![\rho']\!]^{-1}]\right]\right)\,[[\![\sigma]\!]^{-1}]\\
&= \left([\![\kappa']\!]_\sigma^{-1}\,\left[\mathbf{do}\,\mathrm{ret}\,s[[\![W]\!]^{-1}\ldots]\,\mathbf{as}\,[\![G]\!]^{-1}[[\![\rho']\!]^{-1}]\right]\right)\,[[\![\sigma,\sigma']\!]^{-1}]\\
&\mapsto_{\beta\,\mathrm{Ret}} \left([\![\kappa']\!]_\sigma^{-1}\,\left[(s[[\![W]\!]^{-1}]\,\mathbf{as}\,[\![G]\!]^{-1})[[\![\rho']\!]^{-1}]\right]\right)\,[[\![\sigma,\sigma']\!]^{-1}]\\
&= [\![\langle s\,\|\,W\ldots\,\|\,G\rangle\,[\rho'\kappa']\,[\sigma,\sigma']]\!]^{-1}
\end{aligned}$$

(Clos) where clos $F[\rho'\ldots];\bullet = \sigma_\rho(V')$ and $W\ldots;\sigma' = \rho^*(V\ldots)$, by lemmas E.1 and E.2

$$\begin{aligned}
&[\![\langle\langle V'.\,\mathrm{call} \parallel k[V\ldots]\rangle\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\left[\langle [\![V']\!]^{-1}.\,\mathrm{call} \parallel k[[\![V]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&= \left([\![\kappa]\!]_\sigma^{-1}\left[\langle \mathrm{clos}\,[\![F]\!]^{-1}[[\![\rho']\!]^{-1}].\,\mathrm{call} \parallel k[[\![W]\!]^{-1}\ldots]\rangle\right]\right)[[\![\sigma,\sigma']\!]^{-1}]\\
&\mapsto_{\beta\,\mathrm{Clos}} \left([\![\kappa]\!]^{-1}_{\sigma,\sigma'}\left[\langle \lambda[\![F]\!]^{-1} \parallel k[[\![W]\!]^{-1}\ldots]\rangle\,[[\![\rho']\!]^{-1}]\right]\right)[[\![\sigma,\sigma']\!]^{-1}]\\
&= [\![\langle F \parallel k \parallel W\ldots\rangle\,[\rho'\kappa]\,[\sigma,\sigma']]\!]^{-1}
\end{aligned}$$

(Proc)

$$\begin{aligned}
&[\![\langle \mathrm{proc}\,F\rangle\,[\rho\ \mathrm{sub}\,\overline{x}]\,[\overline{x} := \mathrm{enter}\,k[W\ldots\kappa],\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\left[\langle \mathrm{proc}\,[\![F]\!]^{-1}.\,\mathrm{enter} \parallel k[[\![W]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&\mapsto_{\beta\,\mathrm{Proc}} \left([\![\kappa]\!]_\sigma^{-1}\left[\langle \lambda[\![F]\!]^{-1} \parallel k[[\![W]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&= [\![\langle F \parallel k \parallel W\ldots\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}
\end{aligned}$$

(*Fun*)

$$\begin{aligned}
&[\![\langle \{\, k[\Gamma_k] \to M_k{}^{k\in Q}_{\ldots} \parallel k' \parallel W\ldots\,\}\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\left[\langle \lambda\,\{\, k[\Gamma_k] \to [\![M_k]\!]^{-1}{}^{k\in Q}_{\ldots}\,\} \parallel k'[[\![W]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&\mapsto_{\beta\lambda} \left([\![\kappa]\!]_\sigma^{-1}\left[[\![M_{k'}]\!]^{-1}[[\![\rho]\!]^{-1}, [\![W]\!]^{-1}\ldots/\Gamma_{k'}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&= [\![\langle M_{k'}\rangle\,[\rho, (\Gamma_{k'} := W\ldots)\kappa]\,[\sigma]]\!]^{-1}
\end{aligned}$$

(*Match*)

$$\begin{aligned}
&[\![\langle s' \parallel W\ldots \parallel \{\, s[\Gamma_s] \to M_s{}^{s\in P}_{\ldots}\,\}\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\left[(s'[[\![W]\!]^{-1}\ldots]\ \mathbf{as}\ \{\, s[\Gamma_s \to [\![M_s]\!]^{-1}]^{s\in P}_{\ldots}\,\})[[\![\rho]\!]^{-1}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&\mapsto_{\beta\,\mathbf{as}} \left([\![\kappa]\!]_\sigma^{-1}\left[[\![M_{s'}]\!]^{-1}[[\![\rho]\!]^{-1}, [\![W]\!]^{-1}\ldots/\Gamma_{s'}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&= [\![\langle M_{s'}\rangle\,[\rho, (\Gamma_{s'} := W\ldots)\kappa]\,[\sigma]]\!]^{-1}
\end{aligned}$$

($Prim_f$) where $V\ldots;\rho' = Load_f(k, W\ldots)$ follows from assumption 6.3 and the parametricity of reference values in primitive functions (assumption 3.1), which ensures that $f(k[W[[\![\sigma]\!]^{-1}]\ldots]) = (f(k[W\ldots]))[[\![\sigma]\!]^{-1}] = (f(k[V\ldots]))[[\![\rho']\!]^{-1}][[\![\sigma]\!]^{-1}]$ in

$$\begin{aligned}
&[\![\langle f \parallel k \parallel W\ldots\rangle\,[\rho\kappa]\,[\sigma]]\!]^{-1}\\
&= \left([\![\kappa]\!]_\sigma^{-1}\left[\langle \lambda f \parallel k[[\![W]\!]^{-1}\ldots]\rangle\,[[\![\rho]\!]^{-1}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&\mapsto_{Prim_f} \left([\![\kappa]\!]_\sigma^{-1}\left[f(k[W\ldots])\right]\right)[[\![\sigma]\!]^{-1}]\\
&= \left([\![\kappa]\!]_\sigma^{-1}\left[f(k[V\ldots])[AM[\![\rho']\!]^{-1}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&= \left([\![\kappa]\!]_\sigma^{-1}\left[[\![AM[\![f(k[V\ldots])]\!]^{O}_{Sig_f(k)}]\!]^{-1}[AM[\![\rho']\!]^{-1}]\right]\right)[[\![\sigma]\!]^{-1}]\\
&= [\![\langle Prim_f(k[V\ldots])\rangle\,[\rho'\kappa]\,[\sigma]]\!]^{-1}
\end{aligned}$$

($Prim_g$) where $V\ldots; \rho' = Load_g(s, W\ldots)$ follows from assumptions 3.1 and 6.3 as above

$$\begin{aligned}
&[\![\langle s \parallel W\ldots \parallel g\rangle\ [\rho\kappa]\,[\sigma]]\!]^{-1} \\
&= \left([\![\kappa]\!]_\sigma^{-1}\ \left[(s[[\![W]\!]^{-1}\ldots]\ \mathbf{as}\ g)[[\![\rho]\!]^{-1}]\right]\right)\ [[\![\sigma]\!]^{-1}] \\
&\mapsto_{Prim_g} \left([\![\kappa]\!]_\sigma^{-1}\ [g(s[W\ldots])]\right)\ [[\![\sigma]\!]^{-1}] \\
&= \left([\![\kappa]\!]_\sigma^{-1}\ \left[g(s[V\ldots])[AM[\![\rho']\!]^{-1}]\right]\right)\ [[\![\sigma]\!]^{-1}] \\
&= \left([\![\kappa]\!]_\sigma^{-1}\ \left[[\![AM[\![g(s[V\ldots])]\!]^{\mathrm{run}}_{Sig_g(s)}]\!]^{-1}[AM[\![\rho']\!]^{-1}]\right]\right)\ [[\![\sigma]\!]^{-1}] \\
&= [\![\langle Prim_g(s[W\ldots])\rangle\ [\rho'\kappa]\,[\sigma]]\!]^{-1}
\end{aligned}$$

□

For the forward direction, we need to be able to compare a decomposition $E[M]$ of an evaluation context E and a reducible expression M to an arbitrary machine configuration m that translates back to it. For an arbitrary $E[M] \sim m$ where M is reducible, the machine configuration m may not have identified the real redex M yet, and must dig through the remaining part of E to get there.

Lemma E.4 (Refocusing). *For all reducible expressions M, if $E[M] \sim m$, then*

$$m \mapsto^*_{\mathbf{do}\,\mathrm{enter}\,\mathbf{as}\,\lambda}\ c[\rho\kappa]\,[\sigma] \not\mapsto^*_{\mathbf{do}\,\mathrm{enter}\,\mathbf{as}\,\lambda}$$

for some c, ρ, κ, and σ such that exactly one *of the following holds:*

(1) $M = AM[\![c]\!]^{-1}[AM[\![\rho]\!]^{-1}][AM[\![\sigma]\!]^{-1}]$ and $E = AM[\![\kappa]\!]_\sigma^{-1}$.

(2) $c = \langle \mathrm{ret}\, S\rangle$, $\sigma(\kappa(\mathrm{sub})) = \mathbf{do}\, G[\rho'\kappa']$, and

$$\begin{aligned}
M &= \left(\mathbf{do}\,\mathrm{ret}\, AM[\![S]\!]^{-1}[AM[\![\rho]\!]^{-1}]\ \mathbf{as}\ AM[\![G]\!]^{-1}[AM[\![\rho']\!]^{-1}]\right)\ [AM[\![\sigma]\!]^{-1}] \\
E &= AM[\![\kappa']\!]_\sigma^{-1}[AM[\![\sigma]\!]^{-1}]
\end{aligned}$$

(3) $c = \langle \mathrm{proc}\, F\rangle$, $\sigma(\kappa(\mathrm{sub})) = \mathrm{enter}\, k[W\ldots\kappa']$, and

$$\begin{aligned}
M &= \langle \mathrm{proc}\, AM[\![F]\!]^{-1}[AM[\![\rho]\!]^{-1}] \parallel k[W\ldots]\rangle\ [AM[\![\sigma]\!]^{-1}] \\
E &= AM[\![\kappa']\!]_\sigma^{-1}
\end{aligned}$$

Proof. Note that the abstract machine rules are deterministic (up to renaming addresses in the store σ), and the combination of **do**, enter, λ, and **as** reduction of the machine always terminates, so there is always a unique normal form $m' = c[\rho\kappa]\,[\sigma]$ such that $m \mapsto^*_{\mathbf{do}\,\mathrm{enter}\,\mathbf{as}\,\lambda} m' \not\mapsto_{\mathbf{do}\,\mathrm{enter}\,\mathbf{as}\,\lambda}$. From lemma E.3, $E[M] \sim m \mapsto^*_{\mathbf{do}\,\mathrm{enter}\,\mathbf{as}\,\lambda} m'$ implies that $E[M] \sim m' = c[\rho\kappa]\,[\sigma]$. Since this is a **do** enter **as** λ-normal form, we know c is none of $\langle \mathbf{do}\, M'\ \mathbf{as}\ G[\Gamma, O]\rangle$, $\langle\langle M'.\,\mathrm{enter} \parallel K\rangle\rangle$, $\langle S\ \mathbf{as}\ G\rangle$, or $\langle\langle \lambda F \parallel K\rangle\rangle$. Among all the remaining cases,

- If $c = \langle s \parallel W\ldots \parallel G\rangle$ then property (1) holds, where the redex under substitution is
$$AM[\![\langle s \parallel W\ldots \parallel G\rangle]\!]^{-1} = s[W\ldots]\ \mathbf{as}\ AM[\![G]\!]^{-1}$$
- If $c = \langle F \parallel k \parallel W\ldots\rangle$ then property (1) holds, where the redex under substitution is
$$AM[\![\langle F \parallel k \parallel W\ldots\rangle]\!]^{-1} = \langle \lambda AM[\![F]\!]^{-1} \parallel k[W\ldots]\rangle$$
- If $c = \langle \mathrm{ret}\, S\rangle$, then property (2) holds, since we know M is a redex, which must have the form $\mathbf{do}\,\mathrm{ret}\, S'\ \mathbf{as}\ G'$ by the definition of $AM[\![\_]\!]^{-1}$.
- If $c = \langle \mathrm{proc}\, F\rangle$, then property (3) holds, since we know M is a redex, which must have the form $\langle \mathrm{proc}\, F.\,\mathrm{enter} \parallel K\rangle$ by the definition of $AM[\![\_]\!]^{-1}$.
- In all other cases, $c = \langle M'\rangle$ and property (1) holds, where the redex under substitution is $AM[\![\langle M'\rangle]\!]^{-1} = AM[\![M']\!]^{-1}$.

□

Lemma E.5 (Forward simulation). *If $M \sim m$ and $M \mapsto M'$ then $m \mapsto^*_{\mathbf{do}\,\mathrm{enter}\,\mathbf{as}\,\lambda} \mapsto m'$ such that $M' \sim m'$.*

Proof. First, decompose the step as $M = E[M_1] \mapsto E[M_1'] = M'$ where M_1 the reducible sub-expression of M that $M_1 \mapsto M_1'$ by directly applying one of the reduction rules. By lemma E.4, we know that $m \mapsto^*_{\mathbf{do}\,\mathrm{enter}\,\mathbf{as}\,\lambda} c[\rho\kappa][\sigma] \not\mapsto^*_{\mathbf{do}\,\mathrm{enter}\,\mathbf{as}\,\lambda}$ with exactly one of the following properties:

(1) $M = AM[\![c]\!]^{-1}[AM[\![\rho]\!]^{-1}, AM[\![\sigma]\!]^{-1}]$ and $E = AM[\![\kappa]\!]_\sigma^{-1}$.
(2) $c = \langle \mathrm{ret}\, S\rangle$, $\sigma(\kappa(\mathrm{sub})) = \mathbf{do}\, G[\rho'\kappa']$, and

$$M = \left(\mathbf{do}\,\mathrm{ret}\, AM[\![S]\!]^{-1}[AM[\![\rho]\!]^{-1}]\ \mathbf{as}\ AM[\![G]\!]^{-1}[AM[\![\rho']\!]^{-1}]\right)[AM[\![\sigma]\!]^{-1}]$$
$$E = AM[\![\kappa']\!]_\sigma^{-1}$$

(3) $c = \langle \mathrm{proc}\, F\rangle$, $\sigma(\kappa(\mathrm{sub})) = \mathrm{enter}\, k[W\ldots\kappa']$, and

$$M = \langle \mathrm{proc}\, AM[\![F]\!]^{-1}[AM[\![\rho]\!]^{-1}] \parallel k[W\ldots]\rangle\ [AM[\![\sigma]\!]^{-1}]$$
$$E = AM[\![\kappa']\!]_\sigma^{-1}$$

We may now proceed to show the reduction step $c[\rho\kappa][\sigma] \mapsto m'$ inversion on the possible combinations of reduction rule of $M_1 \mapsto M_1'$ and property (1-3) from above:

(β Box) Property (1) and $M_1 = \mathbf{unbox}\,\mathrm{box}\, S\,\mathbf{as}\, G \mapsto S\,\mathbf{as}\, G$. There are two possibilities, depending on if $\mathrm{box}\, S$ has already been stored in σ, or is directly in the code of c, where $G = AM[\![G']\!]^{-1}[AM[\![\rho]\!]^{-1}, AM[\![\sigma]\!]^{-1}]$:
– $\mathrm{box}\, S$ is stored in σ at $\mathrm{ref}\, x$: so $\sigma(\rho(\mathrm{ref}\, x)) = \mathrm{box}\, s[W\ldots]$ in

$$\langle \mathbf{unbox}\,\mathrm{ref}\, x\,\mathbf{as}\, G'\rangle\ [\rho\kappa][\sigma] \mapsto_{\mathrm{Box}} \langle s \parallel W\ldots \parallel G\rangle\ [\rho\kappa][\sigma]$$

which is similar to $E[S\,\mathbf{as}\,G]$ by property (1).
– $\mathrm{box}\, S$ is in the code of c: suppose $W\ldots; \sigma' = \rho^*(V\ldots)$ in

$$\langle \mathbf{unbox}\,\mathrm{box}\, sV\ldots\,\mathbf{as}\, G'\rangle\ [\rho\kappa][\sigma] \mapsto_{\mathrm{Box}} \langle s \parallel W\ldots \parallel G\rangle\ [\rho\kappa][\sigma, \sigma']$$

which is similar to $E[S\,\mathbf{as}\,G]$ by property (1) and lemma E.1.

(β Clos) Property (1) and $M_1 = \langle \mathrm{clos}\, F.\,\mathrm{call} \parallel K\rangle \mapsto \langle \lambda F \parallel K\rangle$. There are two possibilities, depending on if $\mathrm{clos}\, F$ has already been stored in σ, or is directly in the code of c, where $F = AM[\![F']\!]^{-1}[AM[\![\rho'']\!]^{-1}, AM[\![\sigma]\!]^{-1}]$:
– $\mathrm{clos}\, F$ is stored in σ at $\mathrm{ref}\, x$: so $\sigma(\rho(\mathrm{ref}\, x)) = z := \mathrm{clos}\, F'[\rho']$ and suppose $W\ldots; \sigma' = \rho^*(V\ldots)$ in

$$\langle\langle \mathrm{ref}\, x.\,\mathrm{call} \parallel k[V\ldots]\rangle\rangle\ [\rho\kappa][\sigma] \mapsto_{\mathrm{Clos}} \langle F \parallel k \parallel W\ldots\rangle\ [\rho'\kappa][\sigma, \sigma']$$

– $\mathrm{clos}\, F'[\Gamma]$ is in the code of c: suppose $W\ldots; \sigma' = \rho^*(V\ldots)$ in

$$\langle\langle \mathrm{clos}\, F'[\Gamma].\,\mathrm{call} \parallel k[V\ldots]\rangle\rangle\ [\rho\kappa][\sigma] \mapsto_{\mathrm{Clos}} \langle F' \parallel k \parallel W\ldots\rangle\ [\rho|_\Gamma\kappa][\sigma, \sigma']$$

Both reducts are similar to $E[\langle \lambda F \parallel K\rangle]$ by property (1) and lemma E.1.

($Prim_g$) Property (1) and $M_1 = S\,\mathbf{as}\, g \mapsto g(S)$. Since we are at an **as**-normal form, the machine configuration must be, for some $S = s[W[AM[\![\sigma]\!]^{-1}]\ldots]$

$$\langle s \parallel W\ldots \parallel g\rangle\ [\rho\kappa][\sigma] \mapsto_{Prim_g} \langle Prim_g(s[V\ldots])\rangle\ [\rho'\kappa][\sigma]$$

where $V\ldots; \rho' = Load_g(s, W\ldots)$, is similar to $E[g(S)]$, by assumptions 3.1 and 6.3:

$$\begin{aligned}
&AM[\![\langle Prim_g(s[V\ldots])\rangle\ [\rho'\kappa]\,[\sigma]]\!]^{-1}\\
&= \left(AM[\![\kappa]\!]_\sigma^{-1}\ \left[AM[\![Prim_g(s[V\ldots])]\!]^{-1}[AM[\![\rho']\!]^{-1}]\right]\right)\ [AM[\![\sigma]\!]^{-1}]\\
&= (AM[\![\kappa]\!]_\sigma^{-1}[AM[\![\sigma]\!]^{-1}])\ \left[AM[\![Prim_g(s[V\ldots])]\!]^{-1}[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\right]\\
&= E\left[AM[\![AM[\![g(s[V\ldots])]\!]^{\mathrm{run}}_{Sig_g(s)}]\!]^{-1}[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\right]\\
&= E\left[g(s[V\ldots])[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\right]\\
&= E\left[g(s[V\ldots[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]])\right]\\
&= E\left[g(s[W\ldots[AM[\![\sigma]\!]^{-1}]])\right]\\
&= E\,[g(S)]
\end{aligned}$$

($Prim_f$) Property (1) and $M_1 = \langle \lambda f \parallel K\rangle \mapsto f(K)$. Since we are at a λ-normal form, the machine configuration must be, for some $K = k[W[AM[\![\sigma]\!]^{-1}]\ldots]$

$$\langle f \parallel k \parallel W\ldots\rangle\ [\rho\kappa]\,[\sigma] \mapsto_{Prim_f} \langle Prim_f(k[V\ldots])\rangle\ [\rho'\kappa]\,[\sigma]$$

where $V\ldots; \rho' = Load_f(k, W\ldots)$ is similar to the $E[f(K)]$ by assumptions 3.1 and 6.3:

$$\begin{aligned}
&AM[\![\langle Prim_f(k[V\ldots])]\rangle\ [\rho'\kappa]\,[\sigma]]\!]^{-1}\\
&= \left(AM[\![\kappa]\!]_\sigma^{-1}\ \left[AM[\![Prim_f(k[V\ldots])]\!]^{-1}[AM[\![\rho']\!]^{-1}]\right]\right)\ [AM[\![\sigma]\!]^{-1}]\\
&= (AM[\![\kappa]\!]_\sigma^{-1}[AM[\![\sigma]\!]^{-1}])\ \left[AM[\![Prim_f(k[V\ldots])]\!]^{-1}[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\right]\\
&= E\left[AM[\![AM[\![f(k[V\ldots])]\!]^{O}_{Sig_f(k)}]\!]^{-1}[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\right]\\
&= E\left[f(k[V\ldots])[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\right]\\
&= E\left[f(k[V[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\ldots])\right]\\
&= E\left[f(k[W[AM[\![\sigma]\!]^{-1}]\ldots])\right]\\
&= E\,[f(K)]
\end{aligned}$$

(β **as**) Property (1) and $M = s'[V\ldots]$ **as** $\{\, s[\Gamma] \to M_s{}^{s\in P}_{\ldots}\,\} \mapsto M_{s'}[V/\Gamma]$. Since we are at an **as**-normal form, the machine configuration must be, for some $W[AM[\![\sigma]\!]^{-1}]\ldots = V[AM[\![\rho]\!]^{-1}]\ldots$ and $M'_s[AM[\![\rho]\!]^{-1}][AM[\![\sigma]\!]^{-1}] = M_s$ for each $s \in P$,

$$\langle s' \parallel W\ldots \parallel \{\, s[\Gamma_s] \to M'_s{}^{s\in P}_{\ldots}\,\}\rangle\ [\rho\kappa]\,[\sigma] \mapsto_{Match} \langle M'_{s'}\rangle\ [\rho, \Gamma_s := W\ldots\kappa]\sigma$$

which is similar to $E[M_{s'}]$ by property (1).

($\beta\lambda$) Property (1) and $M = \langle \lambda\, \{\, k[\Gamma] \to M_s{}^{k\in Q}_{\ldots}\,\} \parallel k'[V\ldots]\rangle \mapsto M_{k'}[V/\Gamma]$. Since we are at a λ-normal form, the machine configuration must be, for some $W[AM[\![\sigma]\!]^{-1}]\ldots = V[AM[\![\rho]\!]^{-1}]\ldots$ and $M'_k[AM[\![\rho]\!]^{-1}][AM[\![\sigma]\!]^{-1}] = M_k$ for each $k \in Q$,

$$\langle \{\, k[\Gamma_s] \to M'_k{}^{k\in Q}_{\ldots}\,\} \parallel k' \parallel W\ldots\rangle\ [\rho\kappa]\,[\sigma] \mapsto_{Fun} \langle M'_{k'}\rangle\ [\rho, \Gamma_k := W\ldots\kappa]\sigma$$

which is similar to $E[M_{k'}]$ by property (1).

(β Ret) Property (2) and $M_1 = \mathbf{do}\ \mathrm{ret}\, s[V\ldots]$ **as** $G \mapsto s[V\ldots]$ **as** G. The machine configuration must be, for some $G = AM[\![G']\!]^{-1}[AM[\![\rho']\!]^{-1}, AM[\![\sigma]\!]^{-1}]$,

$$\langle \mathrm{ret}\, s[V\ldots]\rangle\ [\rho\ \mathrm{sub}\ \overline{x}]\,[\overline{x} := \mathbf{do}\ G'[\rho'\kappa'], \sigma] \mapsto_{\mathrm{Ret}} \langle s \parallel W\ldots \parallel G'\rangle\ [\rho'\kappa']\,[\sigma]$$

because $W\ldots; \sigma' = \rho^*(V\ldots)$, which is similar to $E[s[V\ldots]\ \mathbf{as}\ G]$ by property (2).

(β Proc) Property (3) and $M_1 = \langle \text{proc } F.\ \text{enter} \parallel k[V\ldots]\rangle \mapsto \langle \lambda F \parallel k[V\ldots]\rangle$. The machine configuration must be, for some $F = AM[\![F']\!]^{-1}[AM[\![\rho]\!]^{-1}, AM[\![\sigma]\!]^{-1}]$,

$$\langle \text{proc } F'\rangle\ [\rho \text{ sub } \overline{x}]\,[\overline{x} := \text{enter } k[W\ldots\kappa'], \sigma] \mapsto_{\text{Proc}} \langle F' \parallel k \parallel W\ldots\rangle\ [\rho\kappa']\,[\sigma]$$

which is similar to $E[\langle \lambda F \parallel k[V\ldots]\rangle]$ by property (3). □

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

(1) For all closed M, $M \sim \langle AM[\![M]\!]^{\text{run}}_{\bullet}\rangle\ [\text{run}]\,[\bullet]$,
(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 $m \mapsto^*_{\textbf{do enter as } \lambda} m'$ 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 $M' \sim m'$.

Proof. (1) Follows by definition of $AM[\![\langle AM[\![M]\!]^{\text{run}}_{\bullet}\rangle\ [\text{run}]\,[\bullet]]\!]^{-1}$ with the fact that, for any term $AM[\![AM[\![M]\!]^{O}_{\Gamma}]\!]^{-1} = M$ since the round-trip just erases the added annotations.

(2) Follows by definition of $\sim$. The non-cyclic requirement of the store σ ensures that there is a well-founded dependency ordering so that $AM[\![\sigma]\!]^{-1}$ is defined and gives a substitution.

(3) If $m \mapsto^*_{\textbf{do enter as } \lambda} m'$ terminal, then we know $M \sim m'$ as well by lemma E.4, and there are two cases for why M terminal depending on the reason why m' terminal.

- Given $m' = \langle f \parallel k \parallel W\ldots\rangle\ [\rho\kappa]\,[\sigma]$ terminal because $V\ldots; \rho' = Load_f(k, W\ldots)$ and $f(k[V\ldots]) =$ terminal, then from the definition of $M \sim m'$ and assumptions 3.1 and 6.3

$$\begin{aligned} M' &= AM[\![\langle f \parallel k \parallel W\ldots\rangle\ [\rho\kappa]\,[\sigma]]\!]^{-1} \\ &= \left(AM[\![\kappa]\!]^{-1}_{\sigma}\ [\langle \lambda f \parallel k[W\ldots]\rangle]\right)\ [AM[\![\sigma]\!]^{-1}] \\ &= \left(AM[\![\kappa]\!]^{-1}_{\sigma}\ \left[\langle \lambda f \parallel k[V[AM[\![\rho']\!]^{-1}]\ldots]\rangle\right]\right)\ [AM[\![\sigma]\!]^{-1}] \\ &= \left(AM[\![\kappa]\!]^{-1}_{\sigma}\ [\langle \lambda f \parallel k[V\ldots]\rangle]\right)\ [AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}] \text{ terminal} \end{aligned}$$

- Given $m' = \langle s \parallel W\ldots \parallel g\rangle\ [\rho \text{ run}]\,[\sigma]$ terminal because $V\ldots; \rho' = Load_f(k, W\ldots)$ and $g(s[V\ldots]) =$ terminal, then from the definition of $M \sim m'$ and assumptions 3.1 and 6.3

$$\begin{aligned} M' &= AM[\![\langle s \parallel W\ldots \parallel g\rangle\ [\rho \text{ run}]\,[\sigma]]\!]^{-1} \\ &= (s[W\ldots]\ \textbf{as}\ g)\ [AM[\![\sigma]\!]^{-1}] \\ &= \left(s[V[AM[\![\rho']\!]^{-1}]\ldots]\ \textbf{as}\ g\right)\ [AM[\![\sigma]\!]^{-1}] \\ &= (s[V\ldots]\ \textbf{as}\ g)\ [AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}] \text{ terminal} \end{aligned}$$

In the other direction, if M terminal, then we can calculate the **do** enter **as** λ-normal form $m \mapsto^*_{\textbf{do enter as } \lambda} m' \not\mapsto_{\textbf{do enter as } \lambda}$ and still have $M \sim m'$ by lemma E.4. From there, there are two cases for why $m' = c[\rho\kappa]\,[\sigma]$ terminal. depending on the reason why M terminal.

- Given M terminal because $M = E[\langle \lambda f \parallel k[V\ldots]\rangle]$ and $f(k[V\ldots]) =$ terminal, then because $\langle \lambda f \parallel k[V\ldots]\rangle$ is a (potentially) reducible expression and m' is a **do** enter **as** λ-normal form, we know that property (1) of lemma E.4

$$\begin{aligned} \langle \lambda f \parallel k[V\ldots]\rangle &= AM[\![c]\!]^{-1}[AM[\![\rho]\!]^{-1}][AM[\![\sigma]\!]^{-1}] \\ E &= AM[\![\kappa]\!]^{-1}_{\sigma} \end{aligned}$$

must hold because (2) and (3) cannot match the reducible expression $\langle \lambda f \parallel k[V\ldots]\rangle$. By inversion on the first equation, and the fact that m' is a λ-normal form, it must be that

$$c = \langle f \parallel k \parallel W\ldots\rangle$$

for some $W\ldots$ such that $V\ldots = W[AM[\![\sigma]\!]^{-1}]\ldots$. It follows from assumptions 3.1 and 6.3. that, for any $V'\ldots; \rho' = Load_f(k, W\ldots)$

$$V'[AM[\![\rho']\!]^{-1}]\ldots = W\ldots$$
$$V'[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\ldots = V\ldots = W[AM[\![\sigma]\!]^{-1}]\ldots$$
$$f(k[V'\ldots]) = f(k[V\ldots]) = \text{terminal}$$

- Given M terminal because $M = s[V\ldots]\ \mathbf{as}\ g$ and $g(s[V\ldots]) = \text{terminal}$, then because $s[V\ldots]\ \mathbf{as}\ g$ is a (potentially) reducible expression and m' is a **do** enter **as** λ-normal form, we know that property (1) of lemma E.4

$$s[V\ldots]\ \mathbf{as}\ g = AM[\![c]\!]^{-1}[AM[\![\rho]\!]^{-1}][AM[\![\sigma]\!]^{-1}]$$
$$\Box = AM[\![\kappa]\!]^{-1}_{\sigma}$$

must hold because (2) and (3) cannot match the reducible expression $s[V\ldots]\ \mathbf{as}\ g$. By inversion these two equations, and the fact that m' is a λ-normal form, it must be that

$$c = \langle s \parallel W\ldots \parallel g\rangle$$
$$\kappa = \text{run}$$

for some $W\ldots$ such that $V\ldots = W[AM[\![\sigma]\!]^{-1}]\ldots$. It follows from assumptions 3.1 and 6.3. that, for any $V'\ldots; \rho' = Load_f(k, W\ldots)$

$$V'[AM[\![\rho']\!]^{-1}]\ldots = W\ldots$$
$$V'[AM[\![\rho']\!]^{-1}][AM[\![\sigma]\!]^{-1}]\ldots = V\ldots = W[AM[\![\sigma]\!]^{-1}]\ldots$$
$$g(s[V'\ldots]) = g(s[V\ldots]) = \text{terminal}$$

Thus, in both cases, m' terminal.

(4) Follows from lemma E.5 by induction on the length of the reduction sequence $M \mapsto^* M'$.

(5) Follows from lemma E.3 by induction on the length of the reduction sequence $m \mapsto^* m'$. □

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

Proof. Follows directly from lemma 6.4. From (1), we know the initial $M \sim \langle AM[\![M]\!]^{\text{run}}_{\bullet}\rangle\ [\text{run}]\ [\bullet]$. Then, if $\langle AM[\![M]\!]^{\text{run}}_{\bullet}\rangle\ [\text{run}]\ [\bullet] \mapsto m'$ terminal, (5) gives $M \mapsto M'$ such that $M' \sim m'$, and (3) ensures M' terminal.

Going the other way, if $M \mapsto^* M'$ terminal, (4) gives $\langle AM[\![M]\!]^{\text{run}}_{\bullet}\rangle\ [\text{run}]\ [\bullet] \mapsto m''$ such that $M' \sim m''$. From M' terminal, (3) ensures $m'' \mapsto^*_{\mathbf{do}\ \text{enter}\ \mathbf{as}\ \lambda} m'$ terminal so that lemma E.4 gives the required $M' \sim m'$. □

F PROOF OF MACHINE TYPE PRESERVATION AND SAFETY

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

Proof. Type preservation property (2) follows almost directly from induction on the derivation of $\Gamma \vdash M : B : O\ \mathbf{comp}$. The only change in the typing rules are in Clos I and Ret E, which restrict the environment of the contained function F or matching G code to only the free variables of F or G respectively. Typability within the restricted environment is a consequence of well-typed substitution (lemma B.1) and the fact that substitution for unused variables is the identity operation.

Type preservation property (1) comes from the following preservation properties for each other category of machine syntax:

- If $\Gamma \vdash M : \Phi$ then $\Gamma \vdash AM[\![M]\!]^{-1} : \Phi$, and similar for values, structures, stacks, *etc.*
- If $\Psi \vdash W : A : R$ **val** then $AM[\![\Psi]\!]^{-1} \vdash W : A : R$ **val**.
- If $\Psi \vdash W\ldots : \Delta$ then $AM[\![\Psi]\!]^{-1} \vdash W\ldots : \Delta$.
- If $\Psi \vdash \rho : \Gamma$ then $AM[\![\Psi]\!]^{-1} \vdash \rho(R\ x) : A[AM[\![\rho]\!]^{-1}] : R$ **val** for every $R\ x : A$ bound by ρ.
- If $\Psi \vdash H : A$ then $AM[\![\Psi]\!]^{-1} \vdash AM[\![H]\!]^{-1} : A :$ ref **val**.
- If $\kappa : \Phi \vdash \Xi$ and $\sigma : (\Psi \dashv \Xi)$ then $AM[\![\Psi]\!]^{-1} \vdash AM[\![\kappa]\!]_\sigma^{-1}[M] :$ void : run **comp** for every $AM[\![\Psi]\!]^{-1} \vdash M : \Phi$.
- If $\Psi \mid E : B \vdash \Xi$ and $\sigma : (\Psi \dashv \Xi)$ then $AM[\![\Psi]\!]^{-1} \vdash AM[\![E]\!]_\sigma^{-1}[M] :$ void : run **comp** for every $AM[\![\Psi]\!]^{-1} \vdash M : B :$ sub **comp**.
- If $\sigma : (\Xi \dashv \Psi)$ then $AM[\![\Psi]\!]^{-1} \vdash AM[\![\sigma(x)]\!]^{-1} : A :$ ref **val** for all $x : A \in \Psi$.
- If $\sigma : (\Xi \dashv \overline{x} : B)$ then $AM[\![\Psi]\!]^{-1} \vdash AM[\![\sigma(\overline{x})]\!]^{-1}[M] :$ void : run **comp** for every atomic computation $AM[\![\Psi]\!]^{-1} \vdash M : B :$ sub **comp**.
- If $\Psi \mid \Gamma \vdash c : \Phi$ then $AM[\![\Psi]\!]^{-1}, \Gamma \vdash AM[\![c]\!]^{-1} : \Phi$.
- If $c[\rho\kappa] : \Phi \vdash \Xi$ and $\sigma : (\Phi \dashv \Xi)$ then $AM[\![\Phi]\!]^{-1} \vdash AM[\![c[\rho\kappa]]\!]_\sigma^{-1} :$ void : run **comp**.
- If m OK then $\bullet \vdash AM[\![m]\!]^{-1} :$ void : run **comp**.

with the following decompilation of store and $AM[\![\Psi]\!]^{-1} = \Gamma$ stack $AM[\![\Xi]\!]^{-1} = \Phi$ environments:

$$AM[\![x : A\ldots]\!]^{-1} = x : A :\ \text{ref}\ \textbf{val}\ \ldots$$

$$AM[\![\Xi]\!]^{-1} = \begin{cases} \text{void} : \text{run}\ \textbf{comp} & (\text{if } \Xi = \bullet) \\ B : \text{sub}\ \textbf{comp} & (\text{if } \Xi = \overline{x} : B) \end{cases}$$

These facts follow by mutual induction on the given typing derivation:

- Cases for M, V, F, G, *etc.*almost always follow directly from the inductive hypothesis since all typing rules are exactly the same, except for Clos I and Ret E. These two cases also require weakening their premise $\Gamma|_\Delta \vdash F : Q$; and $\Gamma|_\Delta$; $G : P \vdash B : O$ **comp** to add back the unused variables that were remove by the Δ restriction as $\Gamma \vdash F : Q$; and Γ ; $G : P \vdash B : O$ **comp**. Weakening a typing derivation to add extra unused variables to the environment follows by induction, as usual.
- The rules for typing register values $\Psi \vdash W : A : R$ **val** are all special cases of the source typing rules for values, but where the environment is restricted to only reference variables. Note that reference variables can never be used in types T (a type can only refer to ty variables via *TyVar*), so $AM[\![\Phi]\!]^{-1} \vdash T : \tau$ if and only if $\bullet \vdash T : \tau$.
- The rules for typing register value sequences $\Psi \vdash W\ldots : \Delta$ are likewise all special cases of value sequences.
- The rules for named registers $\Psi \vdash \rho : \Gamma$ binds a W to each variable named by Γ, but note that the types of subsequent bindings may refer to previous ones. For example, we may have

  $$\bullet \vdash (\text{ty}\ \ a := \text{Nat}, \text{int}\ \ x := 3) : (a : \text{Type int}\ \textbf{val} : \text{ty}\ \textbf{val}, x : a : \text{int}\ \textbf{val}).$$

  This is why we need to close each resulting type by substituting ρ. In this example where $\rho = \text{ty}\ \ a := \text{Nat}, \text{int}\ \ x := 3$, we have:

  $$\bullet \vdash\ \ \rho(\text{ty}\ a) : \text{Type int}\ \textbf{val}\ [AM[\![\rho]\!]^{-1}] : \text{ty}\ \textbf{val} \quad \equiv \quad \bullet \vdash \text{Nat} : \text{Type int}\ \textbf{val} : \text{ty}\ \textbf{val}$$

  $$\bullet \vdash \rho(\text{int}\ x) : a[AM[\![\rho]\!]^{-1}] \quad : \text{int}\ \textbf{val} \quad \equiv \quad \bullet \vdash \quad 3 : \text{Nat} \quad : \text{int}\ \textbf{val}$$

- The rules for heap objects $\Psi \vdash H : A$ correspond to the introduction rules Box I and Clos I where the values in the boxed structure are limited to register values W, and the function code F in the closure has a register assignment ρ for all non-reference free variables of F. Types are preserved since the decompilation $AM[\![\text{clos}\ F[\rho]]\!]^{-1} = \text{clos}\ AM[\![F]\!]^{-1}[AM[\![\rho]\!]^{-1}]$ closes all of F's non-reference variables via substitution.

- The rules for stack frames $\Psi \mid E : B \vdash \Xi$ and registers $\kappa : \Phi \vdash \Xi$ are the trickiest, since they require a store to decompile as $AM[\![E]\!]_\sigma^{-1}$ and $AM[\![\kappa]\!]_\sigma^{-1}$ into an evaluation context, which then needs to be filled by an appropriately typed computation. In these cases, the shrinking inductive argument is the stack pointer $\overline{x}$ into σ, which will move closer to the end at each step until it finally reaches the bottom of the stack where $\kappa = \mathrm{run}$. These cases are:
  - Suppose
$$\frac{}{\mathrm{run} : \mathrm{void} : \mathrm{run}\ \mathbf{comp} \vdash \bullet}$$
    Then the conclusion is immediate.
  - Suppose $\sigma : (\Psi \dashv \overline{x} : B)$ and
$$\frac{}{\mathrm{sub}\ \overline{x} : B : \mathrm{sub}\ \mathbf{comp} \vdash \overline{x} : B}$$
    Then $AM[\![\mathrm{sub}\ \overline{x}]\!]_\sigma^{-1} = AM[\![\sigma(\overline{x})]\!]_\sigma^{-1}$ and the result follows by the inductive hypothesis on σ.
  - Suppose $\sigma : (\Psi \dashv \Xi)$ and
$$\frac{\bullet \mid \Delta\ ;\ k : Q \vdash \Phi \quad \Psi \vdash W\ldots : \Delta \quad \Psi \mid \kappa : \Phi \vdash \Xi}{\Psi \mid \mathrm{enter}\ k[W\ldots\kappa] : \mathrm{Proc}\ Q \vdash \Xi}$$
    The inductive hypotheses ensure
$$\begin{aligned} &AM[\![\Psi]\!]^{-1} \vdash W\ldots &&: \Delta \\ &AM[\![\Psi]\!]^{-1} \vdash AM[\![\kappa]\!]_\sigma^{-1}[M] : \mathrm{void} : \mathrm{run}\ \mathbf{comp} && (\forall AM[\![\Psi]\!]^{-1} \vdash M : \Phi) \end{aligned}$$
    Observe that $AM[\![\mathrm{enter}\ k[W\ldots\kappa]]\!]_\sigma^{-1} = AM[\![\kappa]\!]_\sigma^{-1}[\langle \Box.\ \mathrm{enter} \parallel k[W\ldots]\rangle]$ and furthermore $AM[\![\Psi]\!]^{-1} \vdash \langle M.\ \mathrm{enter} \parallel k[W\ldots]\rangle : \Phi$ for all $AM[\![\Psi]\!]^{-1} \vdash M : \mathrm{proc}\ Q : \mathrm{sub}\ \mathbf{comp}$. The result follows by composing this evaluation context with the one from the inductive hypothesis.
  - Suppose $\sigma : (\Psi \vdash \Xi)$ and
$$\frac{\Gamma\ ;\ G : P \vdash \Phi \quad \Psi \vdash \rho : \Gamma \quad \Psi \mid \kappa : \Phi \vdash \Xi}{\Psi \mid \mathbf{do}\ G[\rho\kappa] : \mathrm{Ret}\ P \vdash \Xi}$$
    The inductive hypotheses ensure
$$\begin{aligned} \Gamma \mid\ &AM[\![G]\!]^{-1} &&: P \vdash \Phi \\ &AM[\![\Psi]\!]^{-1} \vdash \rho(R\ x) &&: A[AM[\![\rho]\!]^{-1}] && (\forall R\ x : A \in BV(\rho)) \\ &AM[\![\Psi]\!]^{-1} \vdash AM[\![\kappa]\!]_\sigma^{-1}[M] : \mathrm{void} : \mathrm{run}\ \mathbf{comp} && && (\forall AM[\![\Psi]\!]^{-1} \vdash M : \Phi) \end{aligned}$$
    Observe that $AM[\![\mathbf{do}\ G[\rho\kappa]]\!]_\sigma^{-1} = AM[\![\kappa]\!]_\sigma^{-1}[\mathbf{do}\ \Box\ \mathbf{as}\ AM[\![G]\!]^{-1}[AM[\![\rho]\!]^{-1}]]$ and furthermore we have $AM[\![\Psi]\!]^{-1} \vdash \mathbf{do}\ M\ \mathbf{as}\ AM[\![G]\!]^{-1}[AM[\![\rho]\!]^{-1}] : \Phi$ for all sub-computations $AM[\![\Psi]\!]^{-1} \vdash M : \mathrm{Ret}\ P : \mathrm{sub}\ \mathbf{comp}$ via the well-typed substitution (lemma B.1) from ρ. The result follows by composing this evaluation context with the one from the inductive hypothesis.
- On the right hand, given a well-typed store $(\sigma, x_i := H_i{}^{i \in I}) : (x_i : A_i{}^{i \in I} \dashv \Xi)$ we can extract all well-typed heap objects H_i bound to each $x_i : A_i$ by the inductive hypothesis on $\Psi_i \vdash H_i : A_i$ for some smaller prefix $\Psi_i, \Psi_i' = \Psi$. Note that since each variable x_i is a reference, it cannot appear in any of the types A_i.
  On the left hand, given a well-typed store $((\overline{x} := E, \sigma'), \sigma) : (\Psi', \Psi \dashv \overline{x} : B)$ we have the top stack frame $\Psi' \mid E : B \vdash \Xi$ in the smaller store $\sigma' : (\Psi' \dashv \Xi)$. Looking up the top stack frame in the whole store is $((\overline{x} := E, \sigma'), \sigma)(\overline{x}) = E$. It follows from the inductive hypotheses on E and σ' that $AM[\![\Psi']\!]^{-1} \vdash AM[\![E]\!]_{\sigma'}^{-1}[M] : \mathrm{void} : \mathrm{run}\ \mathbf{comp}$ for all $AM[\![\Psi]\!]^{-1} \vdash M : B : \mathrm{sub}\ \mathbf{comp}$, and thus the result follows by weakening the inductive hypothesis by Ψ.

- The rules for $\Psi \mid \Gamma \vdash c : \Phi$ all follow directly by the definition of $AM[\![c]\!]^{-1}$ and the inductive hypotheses. Likewise the rules for $\Psi \vdash c[\rho\kappa] : \Xi$ and m OK follow by the definitions of $AM[\![c[\rho\kappa]]\!]^{-1}_{\sigma}$ and $AM[\![m]\!]^{-1}$ along with the inductive hypotheses and well-typed substitution (lemma B.1). □

Lemma F.1. *Given any well-typed*

$$\Gamma \vdash V\ldots : \Delta \qquad \Psi \vdash \rho : \Gamma \qquad \sigma : (\Psi \dashv \Xi)$$

there exists a solution to $\rho^*(V) = W\ldots; \sigma'$ *such that*

$$\Psi' \vdash W\ldots : \Delta \qquad \sigma' : (\Psi' \dashv \bullet) \qquad \sigma, \sigma' : (\Psi, \Psi' \dashv \Xi)$$

Proof (sketch). By induction on the typing derivation of the value sequence $\Gamma \vdash V\ldots : \Delta$ and registers $\Psi \vdash \rho : \Gamma$. We need the fact that ρ closes over the Γ used by $V\ldots$ in the cases for variable $R\ x$, type T, and closure $\mathrm{clos}\, F[\Delta]$ values, since $\rho^*(V\ldots, R\ x)$ and $\rho^*(V\ldots, T)$ and $\rho^*(\mathrm{clos}\, F[\Delta])$ will look up one or more values in ρ to create a closed register value $\rho(R\ x)$, a closed type $T[\rho(\mathrm{ty}\, x)/\mathrm{ty}\, x^{x \in FV(T)}_{\ldots}]$, and a function closure $\mathrm{clos}\, F[\rho|_\Delta]$, respectively. In the cases for $\mathrm{box}\, s[V'\ldots]$ and $\mathrm{clos}\, F[\Delta]$ where one or more references have to be allocated in σ', we can always pick fresh addresses that aren't already assigned in the given σ. □

Lemma F.2. *Given any well-typed*

$$\Gamma \vdash V : A : \mathrm{ref}\ \mathbf{val} \qquad \Psi \vdash \rho : \Gamma \qquad \sigma : (\Psi \dashv \Xi)$$

there exists a solution to $\sigma_\rho(V) = H; \sigma'$ *such that*

$$\sigma' : (\Psi' \dashv \bullet) \qquad \sigma, \sigma' : (\Psi, \Psi' \dashv \Xi) \qquad \Psi, \Psi' \vdash H : A$$

Proof (sketch). By induction on the typing derivations of $\Gamma \vdash V : A : \mathrm{ref}\ \mathbf{val}$ and $\Psi \vdash \rho : \Gamma$.

- When $V = \mathrm{ref}\, x$ by the *Var* rule, then $x : A : \mathrm{ref}\ \mathbf{val}$ is in Γ, which forces the binding $\sigma(\rho(\mathrm{ref}\, x)) = H : A$ to exist. In this case, $\sigma' = \bullet$.
- When $V = \mathrm{clos}\, F[\Delta]$ by the Clos I rule, we have $A = \mathrm{Clos}\, Q$ and Δ names a subset of bindings in Γ, so that $H = \mathrm{clos}\, F[\rho|_\Delta] : \mathrm{Clos}\, Q$. In this case, $\sigma' = \bullet$.
- When $V = \mathrm{box}\, s[V'\ldots]$ by the Box I rule, we have $A = \mathrm{Box}\, P$, $\Gamma \mid \Delta \vdash s : P$;, and $\Gamma \vdash V'\ldots : \Delta$. From lemma F.1, we get a solution to $\rho^*(V'\ldots) = W\ldots; \sigma'$ such that

$$\sigma' : (\Psi' \dashv \bullet) \qquad \sigma, \sigma' : (\Psi, \Psi' \dashv \Xi) \qquad \Psi, \Psi' \vdash \mathrm{box}\, sW\ldots : \mathrm{Box}\, P$$ □

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

Proof. We prove both progress and preservation simultaneously by induction on the typing derivation of the $c[\rho\kappa][\sigma]$ OK to show that either $c[\rho\kappa][\sigma] \mapsto m$ OK or $c[\rho\kappa][\sigma]$ terminal.

All of these cases begin with the two cut rules

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

and we proceed as follows for each of the typing derivations of $\Gamma \vdash c : \Phi$:

- $c = \langle s[V\ldots]\ \mathbf{as}\ G\rangle$ is typed by

$$\dfrac{\dfrac{\Gamma \vdash s[V\ldots] : P \;; \quad \Gamma \;; G : P \vdash \Phi}{\Gamma \vdash s[V\ldots]\ \mathbf{as}\ G : \Phi}}{\Psi \mid \Gamma \vdash \langle s[V\ldots]\ \mathbf{as}\ G\rangle : \Phi}$$

Lemma F.1 ensures a well-typed solution $W\ldots;\sigma' = \rho^*(V\ldots)$ such that the machine **as**-steps to an OK new configuration by weakening $\Psi \vdash \rho : \Gamma$

$$\dfrac{\Psi, \Psi' \vdash \rho : \Gamma \qquad \dfrac{\dfrac{\Gamma \mid \Delta \vdash s : P\,; \quad \Psi, \Psi' \vdash W\ldots : \Delta \quad \Gamma\,; G : P \vdash \Phi}{\Psi, \Psi' \mid \Gamma \vdash \langle s \parallel W\ldots \parallel G\rangle : \Phi} \qquad \kappa : \Phi \vdash \Xi}{\Psi, \Psi' \vdash \langle s \parallel W\ldots \parallel G\rangle\,[\rho\kappa] : \Xi} \qquad \sigma, \sigma' : (\Psi, \Psi' \dashv \Xi)}{\langle s \parallel W\ldots \parallel G\rangle\,[\rho\kappa]\,[\sigma, \sigma'] \text{ OK}}$$

- $c = \langle\langle \lambda F \parallel k[V\ldots]\rangle\rangle$ is typed by

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash F : Q\,;}{\Gamma \vdash \lambda F : Q} \quad \dfrac{\Gamma \mid \Delta\,; k \vdash \Phi \quad \Gamma \vdash V\ldots : \Delta}{\Gamma \mid k[V\ldots] : Q \vdash \Phi}\ \textit{Stack}}{\Gamma \vdash \langle \lambda F \parallel k[V\ldots]\rangle : \Phi}\ \textit{StackCut}}{\Psi \mid \Gamma \vdash \langle\langle \lambda F \parallel k[V\ldots]\rangle\rangle : \Phi}$$

  and can always step by λ-reduction to an OK new state analogously to the previous case.
- $c = \langle \mathbf{unbox}\, V\, \mathbf{as}\, G\rangle$ is typed by

$$\dfrac{\dfrac{\Gamma \vdash V : \mathrm{Box}\, P : \mathrm{ref}\ \mathbf{val} \quad \Gamma\,; G : P \vdash \Phi}{\Gamma \vdash \mathbf{unbox}\, V\, \mathbf{as}\, G : \Phi}}{\Psi \mid \Gamma \vdash \langle \mathbf{unbox}\, V\, \mathbf{as}\, G\rangle : \Phi}$$

  Lemma F.2 ensures a well-typed solution $H; \bullet = \sigma_\rho(V)$ such that $\Psi, \Psi' \vdash H : \mathrm{Box}\, P$. It must be that $H = \mathrm{box}\, s[W\ldots]$ with the only possible typing derivation

$$\dfrac{\bullet \mid \Delta \vdash s : P\,; \quad \Psi, \Psi' \vdash W\ldots : \Delta}{\Psi, \Psi' \vdash \mathrm{box}\, s[W\ldots] : \mathrm{Box}\, P}$$

  and so the machine Box-steps to the OK new state by weakening $\Psi \vdash \rho : \Gamma$

$$\dfrac{\Psi, \Psi' \vdash \rho : \Gamma \qquad \dfrac{\dfrac{\Gamma \mid \Delta \vdash s : P\,; \quad \Psi, \Psi' \vdash W\ldots : \Delta \quad \Gamma\,; G : P \vdash \Phi}{\Psi, \Psi' \mid \Gamma \vdash \langle s \parallel W\ldots \parallel G\rangle : \Phi} \qquad \kappa : \Phi \vdash \Xi}{\Psi, \Psi' \vdash \langle s \parallel W\ldots \parallel G\rangle\,[\rho\kappa] : \Xi} \qquad \sigma, \sigma' : (\Psi, \Psi' \dashv \Xi)}{\langle s \parallel W\ldots \parallel G\rangle\,[\rho\kappa]\,[\sigma, \sigma'] \text{ OK}}$$

- $c = \langle\langle V'.\,\mathrm{call} \parallel k[V\ldots, O]\rangle\rangle$ is typed by

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash V' : \mathrm{clos}\, Q : \mathrm{ref}\ \mathbf{val}}{\Gamma \vdash V'.\,\mathrm{call} : Q} \quad \dfrac{\Gamma \mid \Delta\,; k \vdash \Phi \quad \Gamma \vdash V\ldots : \Delta}{\Gamma \mid k[V\ldots] : Q \vdash \Phi}\ \textit{Stack}}{\Gamma \vdash \langle V'.\,\mathrm{call} \parallel k[V\ldots,]\rangle : \Phi}\ \textit{StackCut}}{\Psi \mid \Gamma \vdash \langle\langle V'.\,\mathrm{call} \parallel k[V\ldots]\rangle\rangle : \Phi}$$

  and can step if and only if $\sigma_\rho(V') = \mathrm{clos}\, F[\rho']$ and $\rho^*(V\ldots) = W\ldots;\sigma'$ such that σ, σ' is valid. The reason this well-typed machine always steps is analogous to the previous cases.
- $c = \langle \mathrm{proc}\, F\rangle$ with $\Phi = \mathrm{Proc}\, Q : \mathrm{sub}\ \mathbf{comp}$ is typed by

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash F : Q\,;}{\Gamma \vdash \mathrm{proc}\, F : \mathrm{Proc}\, Q : \mathrm{sub}\ \mathbf{comp}}\ \mathrm{Proc}\, I}{\Psi \mid \Gamma \vdash \langle \mathrm{proc}\, F\rangle : \mathrm{Proc}\, Q : \mathrm{sub}\ \mathbf{comp}}$$

  and can step by Proc-reduction if and only if $\kappa = \mathrm{sub}\, \overline{x}$ and $\sigma = (\overline{x} := \mathrm{enter}\, k[W\ldots\kappa'], \sigma')$. This is ensured because $\Phi = \mathrm{Proc}\, Q : \mathrm{sub}\ \mathbf{comp}$ is forced by the $\mathrm{Proc}\, I$ rule, which forces

$\kappa = \mathrm{sub}\,\overline{x} : \mathrm{Proc}\,Q : \mathrm{sub}\,\mathbf{comp} \vdash \overline{x} : \mathrm{Proc}\,Q = \Xi$, and thus $\sigma : \Psi \vdash \overline{x} : \mathrm{Proc}\,Q$ typed by

$$\dfrac{\dfrac{\bullet \mid \Delta \,;\, k : Q \vdash \Phi' \quad \Psi \vdash W\ldots : \Delta \quad \Psi \mid \kappa' : \Phi' \vdash \Xi}{\Psi \mid \mathrm{enter}\,k[W\ldots\kappa'] : B \vdash \Xi} \qquad \sigma' : (\Psi \vdash \Xi)}{\sigma = (\overline{x} := \mathrm{enter}\,k[W\ldots\kappa'], \sigma') : \Psi \vdash \overline{x} : \mathrm{Proc}\,Q}$$

by reassocating $\overline{x}$ to the top of the store (and weakening enter $k[W\ldots\kappa']$ as necessary) so that the new state is well-typed by the derivation:

$$\dfrac{\Psi \vdash \rho : \Gamma \qquad \dfrac{\dfrac{\Gamma \vdash F : Q \,;\quad \Gamma \mid \Delta \,;\, k : Q \vdash \Phi' \quad \Psi \vdash W\ldots : \Delta}{\Psi \mid \Gamma \vdash \langle F \parallel k \parallel W\ldots\rangle : \Phi'} \qquad \kappa' : \Phi' \vdash \Xi}{\Psi \vdash \langle F \parallel k \parallel W\ldots\rangle\,[\rho\kappa] : \Xi} \qquad \sigma' : (\Psi \dashv \Xi)}{\langle F \parallel k \parallel W\ldots\rangle\,[\rho\kappa'][\sigma']\,\mathrm{OK}}$$

- $c = \langle \mathrm{ret}\,s[V\ldots]\rangle$ with $\Phi = \mathrm{Ret}\,P$ is typed by

$$\dfrac{\dfrac{\dfrac{\Gamma \mid \Delta \vdash s : P \,;\quad \Gamma \vdash V\ldots : \Delta}{\Gamma \vdash s[V\ldots] : P}\,\mathit{Struct}}{\Gamma \vdash \mathrm{ret}\,s[V\ldots] : \mathrm{Ret}\,P}\,\mathrm{Ret}\,I}{\Psi \mid \Gamma \vdash \langle \mathrm{ret}\,s[V\ldots]\rangle : \mathrm{Ret}\,P}$$

  and can step if and only if $\kappa = \mathrm{sub}\,\overline{x}$ and $\sigma = (\overline{x} := \mathbf{do}\,G[\rho'\kappa'], \sigma')$ and $\rho^*(V\ldots) = W\ldots; \sigma''$ such that σ', σ'' is valid. The reason this well-typed machine always steps by Ret-reduction is analogous to the previous cases.
- $c = \langle \mathbf{do}\,M\,\mathbf{as}\,G[\Delta, O]\rangle$ with $\Phi = B : O\,\mathbf{comp}$ is typed by

$$\dfrac{\dfrac{\Gamma \vdash M : \mathrm{Ret}\,P : \mathrm{sub}\,\mathbf{comp} \quad \Gamma|_\Delta \,;\, G : P \vdash B : O\,\mathbf{comp}}{\Gamma \vdash \mathbf{do}\,M\,\mathbf{as}\,G[\Delta, O] : B : O\,\mathbf{comp}}\,\mathrm{Ret}\,E}{\Psi \mid \Gamma \vdash \langle \mathbf{do}\,M\,\mathbf{as}\,G[\Delta, O]\rangle : B : O\,\mathbf{comp}}$$

  and can always step to a well-typed state by **do**-reduction because ρ closes over γ, so $\Psi \vdash \rho : \Gamma$ implies $\Psi \vdash \rho|_\Delta : \Gamma|_\Delta$ in:

$$\mathcal{D} = \dfrac{\Psi \vdash \rho : \Gamma \qquad \dfrac{\Gamma \vdash M : \mathrm{Ret}\,P : \mathrm{sub}\,\mathbf{comp}}{\Psi \mid \Gamma \vdash \langle M\rangle : \mathrm{Ret}\,P : \mathrm{sub}\,\mathbf{comp}} \qquad \mathrm{sub}\,\overline{x} : \mathrm{Ret}\,P : \mathrm{sub}\,\mathbf{comp} \vdash \overline{x} : \mathrm{Ret}\,P}{\Psi \vdash \langle M\rangle\,[\rho\,\mathrm{sub}\,\overline{x}] : \overline{x} : \mathrm{Ret}\,P}$$

$$\mathcal{E} = \dfrac{\dfrac{\Gamma|_\Delta \,;\, G : P \vdash B : O\,\mathbf{comp} \quad \Psi \vdash \rho|_\Delta : \Gamma|_\Delta \quad \Psi \mid \kappa(O) : B : O\,\mathbf{comp} \vdash \Xi}{\Psi \mid \mathbf{do}\,G[\rho|_\Delta\kappa(O)] : \mathrm{Ret}\,P \vdash \Xi} \qquad \sigma : (\Psi \dashv\!: \Xi)}{(\overline{x} := \mathbf{do}\,G[\rho|_\Delta\kappa(O)], \sigma) : (\Psi \dashv \overline{x} : \mathrm{Ret}\,P)}$$

$$\dfrac{\vdots\ \mathcal{D} \qquad\qquad \vdots\ \mathcal{E}}{\Psi \vdash \langle M\rangle\,[\rho\,\mathrm{sub}\,\overline{x}] : \overline{x} : \mathrm{Ret}\,P \quad (\overline{x} := \mathbf{do}\,G[\rho|_\Delta\kappa(O)], \sigma) : (\Psi \dashv \overline{x} : \mathrm{Ret}\,P)}{\langle M\rangle\,[\rho\,\mathrm{sub}\,\overline{x}][\overline{x} := \mathbf{do}\,G[\rho|_\Delta\kappa(O)], \sigma]\,\mathrm{OK}}$$

- $c = \langle\langle M.\,\mathrm{enter} \parallel k[V\ldots, O]\rangle\rangle$ with $\Phi = B : O\,\mathbf{comp}$ is typed by

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash M : \mathrm{Proc}\,Q}{\Gamma \vdash M.\,\mathrm{enter} : Q}\,\mathrm{Proc}\,E \qquad \dfrac{\Gamma \mid \Delta \,;\, k : Q \vdash B : O\,\mathbf{comp} \quad \Gamma \vdash V\ldots : \Delta}{\Gamma \mid k[V\ldots, O] : \mathrm{Proc}\,Q \vdash B : O\,\mathbf{comp}}\,\mathit{Stack}}{\Gamma \vdash \langle M.\,\mathrm{enter} \parallel k[V\ldots, O]\rangle : B : O\,\mathbf{comp}}\,\mathit{StackCut}}{\Psi \mid \Gamma \vdash \langle\langle M.\,\mathrm{enter} \parallel k[V\ldots, O]\rangle\rangle : B : O\,\mathbf{comp}}$$

and can always step to a well-typed state by enter-reduction analogously to the previous case.

- $c = \langle F \parallel k' \parallel W... \rangle$ with $\Phi = \Phi_{k'}$ is typed by

$$\frac{\Gamma \vdash F : Q\ ; \quad \Gamma \mid \Delta'\ ; k' : Q \vdash \Phi_{k'} \quad \Psi \vdash W... : \Delta'}{\Psi \mid \Gamma \vdash \langle F \parallel k' \parallel W... \rangle : \Phi_{k'}}$$

There are one of two cases depending on F.

– F is a copattern-matching function definition typed by

$$\frac{\forall(\Gamma \mid \Delta_k\ ; k : Q \vdash \Phi_k). \quad \Gamma, \Delta_k \vdash M_k : \Phi_k}{\Gamma \vdash \{\, k[\Delta_k] \to M_k \overset{k \in Q}{\ldots} \,\} : Q\ ;}\ \mathit{CoMatch}$$

so that the machine can step by *Fun* reduction since the set of copatterns in the premise covers every possible stack shape of Q, including this chosen k', so we get new OK configuration because $\Psi \vdash \rho, (\Delta_{k'} := W...) : \Gamma, \Delta_{k'}$ by induction on binding pairs between $\Delta_{k'}$ and $W...$:

$$\frac{\dfrac{\Psi \vdash \rho, (\Delta_{k'} := W...) : \Gamma, \Delta_{k'} \quad \dfrac{\Gamma, \Delta_{k'} \vdash M_{k'} : \Phi_{k'}}{\Psi \mid \Gamma, \Delta_{k'} \vdash \langle M_{k'} \rangle : \Phi_{k'}} \quad \kappa : \Phi_{k'} \vdash \Xi}{\Psi \vdash \langle M_{k'} \rangle\, [\rho, (\Delta_{k'} := W...)\kappa] : \Xi} \quad \sigma : (\Psi \dashv \Xi)}{\langle M_{k'} \rangle\, [\rho, (\Delta_{k'} := W...)\kappa]\, [\sigma]\ \mathrm{OK}}$$

– $F = f : Q$ by *PrimFun*. If $f(k'[W...])$ terminal, the machine configuration is terminal. Otherwise, let $V...; \rho' = \mathit{Load}_f(k', W...)$, so that $f(k'[W...]) = f(k'[V...])[AM[\![\rho']\!]^{-1}]$ by assumption 3.1 and has a well-typed result by assumptions 3.3 and 6.6 where:

$$\Psi \vdash \rho' : \mathit{Sig}_f(k') \qquad\qquad \mathit{Sig}_f(k') \vdash f(k'[V...]) : \Phi_{k'}$$

In this case, theorem 6.7 ensures $\mathit{Sig}_f(k') \vdash \mathit{Prim}_f(k'[V...]) : \Phi_{k'}$ and the machine steps by Prim_f to the well-typed configuration:

$$\frac{\Psi \vdash \rho' : \mathit{Sig}_f(k') \quad \dfrac{\mathit{Sig}_f(k') \vdash \mathit{Prim}_f(k'[V...]) : \Phi_{k'}}{\Psi \mid \mathit{Sig}_f(k') \vdash \langle \mathit{Prim}_f(k'[V...]) \rangle : \Phi_{k'}} \quad \kappa : \Phi_{k'}\ \mathbf{comp} \vdash \Xi}{\Psi \vdash \langle \mathit{Prim}_f(k'[V...]) \rangle\, [\rho'\kappa] : \Xi} \quad \sigma : (\Psi \dashv \Xi)}{\langle \mathit{Prim}_f(k'[V...]) \rangle\, [\rho'\kappa]\, [\sigma]\ \mathrm{OK}}$$

- $c = \langle s' \parallel W... \parallel G \rangle$ is typed by

$$\frac{\Gamma \mid \Delta \vdash s' : P\ ; \quad \Psi \vdash W... : \Delta \quad \Gamma\ ; G : P \vdash \Phi}{\Psi \mid \Gamma \vdash \langle s' \parallel W... \parallel G \rangle : \Phi}$$

There are one of two cases depending on G.

– G is a pattern-matching definition typed by

$$\frac{\forall(\Gamma \mid \Delta_s \vdash s : P\ ;). \quad \Gamma, \Delta_s \vdash M_s : \Phi}{\Gamma\ ; \{\, s[\Delta_s] \to M_s \overset{s \in P}{\ldots} \,\} : P \vdash \Phi}\ \mathit{Match}$$

so that the machine can step by *Match* reduction since the set of patterns in the premise covers every possible structure shape of P, including this chosen s', so we get new OK configuration because $\Psi \vdash \rho, (\Delta_{s'} := W...) : \Gamma, \Delta_{s'}$ by induction on binding pairs between

$\Delta_{s'}$ and $W\ldots$:

$$\dfrac{\Psi \vdash (\rho, \Delta_{s'} := W\ldots) : \Gamma, \Delta_{k'} \quad \dfrac{\Gamma, \Delta_{s'} \vdash M_{s'} : \Phi}{\Psi \mid \Gamma, \Delta_{s'} \vdash \langle M_{s'} \rangle : \Phi} \quad \kappa : \Phi \vdash \Xi}{\dfrac{\Psi \vdash \langle M_{s'} \rangle\, [\rho, (\Delta_{s'} := W\ldots)\kappa] : \Xi \qquad \sigma : (\Psi \dashv \Xi)}{\langle M_{s'} \rangle\, [\rho, (\Delta_{s'} := W\ldots)\kappa][\sigma]\ \mathrm{OK}}}$$

– $G = g : P$ by *PrimMatch* where $\Phi = \mathrm{void} : \mathrm{run}\,\mathbf{comp}$. If $g(s'[W\ldots])$ terminal, the machine configuration is terminal. Otherwise, let $V\ldots; \rho' = \mathit{Load}_g(s', W\ldots)$, so that $g(s'[W\ldots]) = g(s'[V\ldots])[AM[\![\rho']\!]^{-1}]$ by assumption 3.1 and has a well-typed defined result by assumptions 3.3 and 6.6 where:

$$\Psi \vdash \rho' : \mathit{Sig}_g(s') \qquad\qquad \mathit{Sig}_g(s') \vdash g(s'[V\ldots]) : \mathrm{void} : \mathrm{run}\,\mathbf{comp}$$

In this case, theorem 6.7 ensures $\mathit{Sig}_g(s') \vdash \mathit{Prim}_g(s'[V\ldots]) : \mathrm{void} : \mathrm{run}\,\mathbf{comp}$ and the machine steps by Prim_g to the well-typed configuration:

$$\dfrac{\Psi \vdash \rho' : \mathit{Sig}_g(s') \quad \dfrac{\mathit{Sig}_f(k') \vdash \mathit{Prim}_f(k'[V\ldots]) : \mathrm{void} : \mathrm{run}\,\mathbf{comp}}{\Psi \mid \mathit{Sig}_g(s') \vdash \langle \mathit{Prim}_g(s'[V\ldots]) \rangle : \mathrm{void} : \mathrm{run}\,\mathbf{comp}} \quad \kappa : \mathrm{void} : \mathrm{run}\,\mathbf{comp} \vdash \Xi}{\dfrac{\Psi \vdash \langle \mathit{Prim}_g(s'[V\ldots]) \rangle\, [\rho'\kappa] : \Xi \qquad \sigma : (\Psi \dashv \Xi)}{\langle \mathit{Prim}_g(s'[V\ldots]) \rangle\, [\rho'\kappa][\sigma]\ \mathrm{OK}}}$$

□

Received 28 February 2024