Kinds Are Calling Conventions

PAUL DOWNEN and ZENA M. ARIOLA, University of Oregon, USA
SIMON PEYTON JONES, Microsoft Research, UK
RICHARD A. EISENBERG, Bryn Mawr College, USA and Tweag I/O, UK

A language supporting polymorphism is a boon to programmers: they can express complex ideas once and reuse functions in a variety of situations. However, polymorphism is pain for compilers tasked with producing efficient code that manipulates concrete values.

This paper presents a new intermediate language that allows for efficient static compilation, while still supporting flexible polymorphism. Specifically, it permits polymorphism over not only the types of values, but also the representation of values, the arity of primitive machine functions, and the evaluation order of arguments—all three of which are useful in practice. The key insight is to encode information about a value's calling convention in the *kind* of its type, rather than in the type itself.

CCS Concepts: • **Software and its engineering** → *Semantics*; *Compilers.*

Additional Key Words and Phrases: arity, levity, representation, polymorphism, type systems

ACM Reference Format:
Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. 2020. Kinds Are Calling Conventions. *Proc. ACM Program. Lang.* 4, ICFP, Article 104 (August 2020), 51 pages. https://doi.org/10.1145/3408986

1 INTRODUCTION

Polymorphism supports reuse by allowing one piece of code to work with values of many different types. But ubiquitous polymorphism usually comes with a runtime cost: all values must share a common representation, usually a pointer to a "boxed" (heap-allocated) object. This is sometimes *much* less efficient than a monomorphic version of the same code, specialized to a particular representation (such as an unboxed 64-bit word).

One approach is to specialize code to a single type. But we would get more reuse if we could specialize to, say, "any type represented by an unboxed 64-bit word." Since *kinds* classify types, perhaps we can write code that is monomorphic in the kind, but polymorphic in the type. Hence our slogan: kinds are calling conventions. For example, consider the function *twice*:

$$twice\ f\ x \ = \ f\ (f\ x)$$

To control performance, we would like to have a say in matters like: Can f be a thunk? How many arguments does f expect (its arity)? Can x be a thunk? How is x represented? Moreover, we want to express the answers to these questions in a type system with strong static guarantees.

A major insight of this paper is the discovery that we can refine the vague notion of "ways in which we want to classify types" along three different axes:

Authors' addresses: Paul Downen; Zena M. Ariola, University of Oregon, Eugene, Oregon, USA, pdownen@cs.uoregon.edu, ariola@cs.uoregon.edu; Simon Peyton Jones, Microsoft Research, Cambridge, UK, simonpj@microsoft.com; Richard A. Eisenberg, Bryn Mawr College, Bryn Mawr, PA, USA , Tweag I/O, Cambridge, UK, rae@richarde.dev.

2475-1421/2020/8-ART104
https://doi.org/10.1145/3408986

- *Representation.* How is this argument represented at runtime?
- *Levity.* What is the evaluation strategy of this argument (*e.g.,* call-by-value or call-by-need)?
- *Arity.* For functions, how many arguments are needed before its code can be executed?

Many functions can be polymorphic in some of these axes, but not in others.

Our focus is on an *intermediate language*. The programmer may write in a uniform language, but the compiler needs an intermediate language that can express low-level representation choices, and expose those choices to the optimizer. For example, the programmer might work exclusively with boxed integer values, say of type `Int`, but the intermediate language can have an unboxed type `Int#`, together with explicit operations to box and unbox integers. This allows the optimizer to eliminate many box-followed-by-unbox chains [Peyton Jones and Launchbury 1991].

This paper builds directly on several earlier works that statically keep track of different representations [Eisenberg and Peyton Jones 2017] and function arities [Downen et al. 2019] within a type and kind system. Our new contribution is to bring them together into a single framework, more powerful and more precise than any of its predecessors. Specifically:

- We introduce a polymorphic intermediate language that statically captures *calling conventions in kinds*, and has polymorphism over the *representation*, *levity*, and *arity* of types (Section 4).
- Our intermediate language is equally well-suited for *both eager and lazy* functional languages. Concretely we show how to compile two higher-level, polymorphic source languages—call-by-name and call-by-value System F—to our intermediate language (Section 5).
- We show how to compile our polymorphic intermediate language to a lower-level language with multiple representations (*e.g.,* pointers versus integers) and multi-arity functions, but *not* polymorphism (Section 6). Compilation is driven by kinds and keeps type erasure and code reuse; typing restrictions ensures polymorphic code is compiled to monomorphic code.
- We provide evidence of correctness for the full compilation process (Theorems 1 to 4) from both call-by-name and call-by-value source languages to machine code.
- We describe a small extension to our intermediate language to allow for dynamic checks on the arities of closures, so that we can use the best arity available at runtime (Section 7).

2 FUNCTION ARITY

We identify three axes of classification, above. Of these three, we peel off arity to explain it first; mixed representations and evaluation strategies have existed for longer and may be more familiar. This section gives a high-level overview of our approach; the details will be nailed down in Section 4.

2.1 What Is Arity?

What is a function's "arity?" In practical terms, the arity of a function determines what code a compiler will generate to call that function. All modern architectures support calling conventions that allow for efficiently calling subroutines by passing multiple arguments at once. If a function has arity n, then a call to that function will pass n arguments simultaneously.

In a naively compiled curried language, every function has arity 1. This implementation of functional languages is unacceptably inefficient. Instead, compilers must somehow map curried surface-language functions onto multi-argument machine-language functions.

To do this, we need two things. First, an *arity analysis*; and second, an *intermediate language* in which arity is explicit, so that we can express and memorialise the results of the analysis.

Arity analysis is not always straightforward. For example:

$$
\begin{array}{ll}
f_1, f_2, f_3 : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} & \\
f_1 = \lambda x.\lambda y.\, \mathbf{let}\, z = \mathit{expensive}\ x\ \mathbf{in}\ y + z & f_2 = \lambda x. f_1\ x \\
f_3 = \lambda x.\, \mathbf{let}\, z = \mathit{expensive}\ x\ \mathbf{in}\ \lambda y. y + z & f_4 = \lambda x. f_3\ x
\end{array}
$$

Here, f_1 clearly has arity 2 because it starts with two λs: it takes two arguments before computing a result. Function f_2 has only one λ, but still has arity 2, because we could safely η-expand it to have two λs. In other words, f_2 has arity 2, just like f_1, because it also requires two arguments before doing serious work. The function f_3 has the same type as f_1, but it *must* take its arguments one at a time: in the call $(map\ (f_3\ 100)\ xs)$ we expect $(expensive\ 100)$ to be computed at most once, whereas the same call for f_1 would recompute $(expensive\ 100)$ for each element of *xs*. Similarly, f_4 has arity 1: it cannot be η-expanded without the risk of computing $(expensive\ 100)$ repeatedly.

These choices become particularly clear in a call-by-value language with side effects. For example, if $(expensive\ 100)$ printed something on the screen, the fact that it is evaluated only once—rather than once for each element of *xs*—is a matter of semantics, not mere efficiency. Even in a pure language like Haskell, an optimizing compiler should still treat computation as a sort of effect; it must, for example, avoid changing the asymptotic efficiency of the program.

In light of these examples, here are two informal definitions of arity:

- An expression e has arity n when it can be soundly η-expanded to $(\lambda x_1 \ldots x_n . e\ x_1 \ldots x_n)$. "Soundness" concerns semantics in an effectful language, but "only" efficiency in a pure one.
- An expression e has arity n when it does no "useful work" until it is applied to n arguments; hence those arguments can be passed simultaneously.

Since the type of a function does not describe its arity (compare f_1 and f_3 above), practical compilers like GHC perform arity analysis based on intensional properties: the form of the expression determines its arity. In its simplest form, we can just count λs. Since that is pessimistic on examples like f_2, GHC uses a variety of simple static heuristic analyses [e.g., Breitner 2014]. The focus of this paper is not on arity analysis, however, but rather on the intermediate language in which we can memorialise the results of that analysis.

2.2 Arity in the Intermediate Language

However arity analysis is done, we need a way to express its results in the intermediate language. In GHC, this is done through an informal decoration on each binder describing its arity.[1] This turns out to be extremely unsatisfactory in practice: GHC has lots and lots of dark corners as a result of this rather squishy notion of arity.

It would be much better if arity were a solid, statically-checked part of the intermediate language. How is that possible? In the world of λ-calculi, we are familiar with calculi having unrestricted β and η rules (such as the call-by-name λ-calculus), as well as calculi having restricted β and η (such as the call-by-value [Sabry and Wadler 1997] or call-by-need λ-calculus [Ariola and Felleisen 1997]). The latter are often used as intermediate languages in compilers, to avoid recomputation of, say, integers. The exact restrictions depend on the language being compiled: its evaluation strategy, whether there are side effects, and so on. Our key idea, introduced by Downen et al. [2019], is this:

> Define an intermediate language ($\mathcal{IL}$) that has *unrestricted η expansion* for functions, while allowing for restricted β reduction on other types of expressions.

This is an unusual choice, so we use a different notation for functions, $\lambda(x{:}\tau).e : \tau \rightsquigarrow \sigma$, where the $(\rightsquigarrow)$ arrow denotes the new primitive function type. Now arity can be read from types: you can freely η-expand *any* expression $e : \tau \rightsquigarrow \sigma$, without reference to the form of e. The previous paper [Downen et al. 2019] and this one are simply working out the consequences of this one idea.

2.3 Currying

If the primitive function type of our intermediate language allows unrestricted η-expansion, how can we express functions like f_3 that intentionally use currying as a way to avoid work duplication?

[1]For aficionados, it is part of the binder's `IdInfo`

We can do so like this:

$$f_3 : \texttt{Int} \rightsquigarrow \{\texttt{Int} \rightsquigarrow \texttt{Int}\}$$
$$f_3 = \lambda x.\,\mathbf{let}\, z = \mathit{expensive}\, x\, \mathbf{in}\, \texttt{Clos}\ (\lambda y.y + z)$$

The type $Int \rightsquigarrow \{Int \rightsquigarrow Int\}$ makes explicit that f_3 is an arity-1 function that returns a closure, denoted by the curly braces, inside which is another arity-1 function. In the term language, λ builds a *primitive function* of type $\tau \rightsquigarrow \sigma$, while `Clos` builds a *closure* of type $\{\tau\}$. Here are the (slightly simplified) introduction (-I) and elimination (-E) rules:

$$\frac{\Gamma, x{:}\tau \vdash e : \sigma}{\Gamma \vdash \lambda x{:}\tau.e : \tau \rightsquigarrow \sigma}\ \textsc{Fun-I} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Clos}\, e : \{\tau\}}\ \textsc{Clo-I}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \sigma}\ \textsc{Fun-E} \qquad \frac{\Gamma \vdash e : \{\tau\}}{\Gamma \vdash \mathsf{App}\ e : \tau}\ \textsc{Clo-E}$$

The App form unboxes the primitive function wrapped up by `Clos`.

Our plan, then, is to desugar the source-language function type $\tau \to \sigma$ into the intermediate-language type as $\{\tau \rightsquigarrow \sigma\}$, adding the corresponding `Clos` and App constructs in the terms.[2] Then we can perform arity analysis, and express its results by transforming the intermediate language program into one with fewer intermediate `Clos` nodes.

2.4 Functions Are Called, Not Evaluated

If we are able to freely η-expand, we must only evaluate functions when they are called. Consider:

$$x = \mathbf{let}\, f : \texttt{Int} \rightsquigarrow \texttt{Int} = \mathit{expensive}\ 100\ \ \mathbf{in}\ \ \ldots f \ldots f \ldots$$

When might *expensive* 100 be evaluated? In a strict language, it is evaluated right away, so the value (some first-class function) can be bound to f before continuing with the body of the **let**. In a lazy language like Haskell, *expensive* 100 might be forced long before f needs to be called, as in $seq\ f\ y$ or with a strict pattern. But in both cases, evaluating f without calling it violates the unrestricted η we desire. After all, the definition of x is η-equivalent to:

$$x' = \mathbf{let}\, f : \texttt{Int} \rightsquigarrow \texttt{Int} = \lambda y.\mathit{expensive}\ 100\ y\ \ \mathbf{in}\ \ \ldots f \ldots f \ldots$$

After η-expansion, the only way to evaluate *expensive* 100 is to *call* f with an argument. Unrestricted η means that x and x' *must* be the same—in both semantics and asymptotic cost.

Unrestricted η-expansion requires a matching evaluation order for function bindings; the evaluator must treat every expression $e : \tau \rightsquigarrow \sigma$ the same as $(\lambda x.e\ x)$. Yet, we do not want to change the evaluation order for expressions of other types, like `Int`. Thus, our language's semantics becomes type-directed: it is only the type of f (in this case, `Int` $\rightsquigarrow$ `Int`) that tells us not to evaluate the right-hand side. In a precise sense (spelled out formally in Section 4), we cannot *evaluate* functions (*e.g.*, due to strictness, *seq*, *etc.*); functions can only be *called*. Furthermore, we will see in Section 3.7 that functions are, in fact, η-expanded during compilation to a lower level. So a semantics that prematurely evaluates functions before they are called is not supported by our machine language.

3 WHY KINDS ARE CALLING CONVENTIONS

So far we have reviewed the ideas of Downen et al. [2019]. We are now ready to introduce the problem we tackle in this paper, and our solution to it.

Previous research asserted *types are calling conventions* [Bolingbroke and Peyton Jones 2009]. We respectfully disagree, instead claiming that *kinds* are calling conventions. As we shall see, this principle offers a unified framework combining several previous works on representations

[2] In Downen et al. [2019], the intermediate language itself had *two* arrows, $(\to)$ as well as $(\rightsquigarrow)$, but we have found it clearer to have only one arrow for primitive functions plus the closure type $\{\tau\}$.

[Eisenberg and Peyton Jones 2017; Peyton Jones and Launchbury 1991], arity [Bolingbroke and Peyton Jones 2009; Downen et al. 2019; Marlow and Peyton Jones 2004], and mixed evaluation strategies [Downen and Ariola 2018] in intermediate languages. Specifically, we provide a type system in which the kind κ of a type $\tau : \kappa$ classifies all the intensional properties needed to compile an expression of type τ, namely its *representation*, *arity*, and *levity*.

3.1 Why Polymorphism Is a Problem

Consider this example of a polymorphic definition, in our intermediate language

$$\begin{aligned}&\mathit{poly} : \forall a. (\texttt{Int} \rightsquigarrow \texttt{Int} \rightsquigarrow a) \rightsquigarrow (a, a)\\ &\mathit{poly} = \Lambda a.\lambda(f : \texttt{Int} \rightsquigarrow \texttt{Int} \rightsquigarrow a).\, \mathbf{let}\, g : \texttt{Int} \rightsquigarrow a = f\ 3\ \mathbf{in}\ (g\ 5, g\ 4)\end{aligned}$$

What is the arity of f and g? With "arities in the types," we count the arrows, and answer 2 and 1 respectively. *But what if a were instantiated by Bool* $\rightsquigarrow$ *Bool?* Then suddenly the answers become 3 and 2 respectively. Yikes!

One solution is to monomorphize the entire program, creating type-specialized versions of each polymorphic function. This is patently unsatisfactory. First, it is a whole-program transformation. Second, some languages (including Haskell and OCaml) support polymorphic recursion, which makes static monomorphization impossible. Third, there is a risk of creating many copies of essentially the same function, many of which wastefully compile to identical machine code.

Instead, we retain the traditional type-erasure model: each polymorphic function is compiled to a single chunk of machine code that works the same no matter how its type is specialized. Under this model, our function *poly* above is problematic. To compile code we must know the arity of every function we call (because its arity determines its calling convention, and thus, what code to generate), but in *poly* we do not know a stable arity of f or g for every instance of a.

3.2 Nonuniform Representations and Polymorphism

Interestingly, this exact same problem has arisen before, in the context of *unboxed data types*. A contribution of this paper is to show that both can be solved with the same idea.

Nearly thirty years ago, GHC introduced the idea of distinguishing boxed and unboxed data types in its intermediate language [Peyton Jones and Launchbury 1991]. They introduced two distinct types for integers: `Int#` for primitive, unboxed machine integers; and `Int` for boxed integers, represented as a pointer to a heap-allocated object and defined as:

$$\mathbf{data}\ \ \texttt{Int} = \texttt{I\#}\ \texttt{Int\#}$$

The (sole) goal of distinguishing these two types is efficiency. If we had only boxed integers, then even simple addition would be forced to evaluate and unbox each argument, then box up the result. By making boxing and unboxing explicit we expose much more low-level information to the optimizer, and can eliminate lots of intermediate boxes. For example, assume we have `plus#` and `minus#` primitive operations, both of type `Int#` $\longrightarrow$ `Int#` $\longrightarrow$ `Int#`, and consider:

$$\begin{aligned}&\mathit{plus}, \mathit{minus} : \texttt{Int} \longrightarrow \texttt{Int} \longrightarrow \texttt{Int}\\ &\mathit{plus}\ \ (\texttt{I\#}\ x)\ (\texttt{I\#}\ y) = \texttt{I\#}\ (\texttt{plus\#}\ x\ y)\\ &\mathit{minus}\ (\texttt{I\#}\ x)\ (\texttt{I\#}\ y) = \texttt{I\#}\ (\texttt{minus\#}\ x\ y)\end{aligned}
\qquad
\begin{aligned}&\mathit{sumFrom} && : \texttt{Int} \longrightarrow \texttt{Int}\\ &\mathit{sumFrom}\ (\texttt{I\#}\ 0) && = \texttt{I\#}\ 0\\ &\mathit{sumFrom}\ x && = \mathit{plus}\, x\ (\mathit{sumFrom}\ (\mathit{minus}\, x\ (\texttt{I\#}\ 1))\end{aligned}$$

This definition is wasteful; each recursive step allocates several new boxes on the heap only to be immediately used by *plus*, *minus*, and *sumFrom*. Instead, as Peyton Jones and Launchbury [1991] show, the recursive function can be optimized using the *worker/wrapper* transformation, like so:

$$\begin{aligned}&\mathit{sumFrom} && : \texttt{Int} \longrightarrow \texttt{Int}\\ &\mathit{sumFrom}\ (\texttt{I\#}\ x) && = \texttt{I\#}\ (\mathit{sumFrom\#}\ x)\end{aligned}
\qquad
\begin{aligned}&\mathit{sumFrom\#} && : \texttt{Int\#} \longrightarrow \texttt{Int\#}\\ &\mathit{sumFrom\#}\ 0 && = 0\\ &\mathit{sumFrom\#}\ x && = \texttt{plus\#}\ x\ (\mathit{sumFrom\#}\ (\texttt{minus\#}\ x\ 1))\end{aligned}$$

Now, the recursion is done by the more efficient *sumFrom*`#` function which works directly on machine integers; no boxes are allocated or consumed, and so *sumFrom*`#` can be compiled with no intermediate allocations. *sumFrom* becomes a wrapper around *sumFrom*`#`, just handling the issues of boxing and unboxing. This is a huge gain in both time and space.

Notice the similarities between this work on unboxed data types and the earlier discussion about arity. In both cases, we make $\mathcal{IL}$ expose more primitive, but less convenient, types of values (primitive functions and unboxed integers respectively), along with a way to explicitly "box" them into a more convenient form (using `Clos` and `I#` respectively) and later "unbox" them (using `App` and pattern matching on `I#` respectively). Making these boxing and unboxing operations explicit in the syntax of $\mathcal{IL}$ programs makes them accessible to the optimizer.

Alas, unboxed types cause trouble with polymorphism. For example, consider the generic, higher-order binary application function:

$$binapp : \forall a\, b\, c.(a \to b \to c) \to a \to b \to c \qquad\qquad binapp\ f\ x\ y = f\ x\ y$$

To compile *binapp* to a single piece of code, we cannot expect the calls *binapp plus* and *binapp* `plus#` to both work; *binapp* would somehow have to handle arguments x and y with different representations (perhaps stored in different registers) to pass them along to *plus* versus `plus#`.

Notice that this is the *exact same problem* that we had with *poly* in Section 3.1: the code we compile for *binapp* depends on how the type variables a and b are instantiated.

3.3 A Stop-Gap Solution

Because of the difficulty with polymorphism, unboxed types were originally introduced with a draconian restriction on polymorphism: *polymorphic type variables (like a, b, c in binapp) cannot be instantiated with unboxed types (like* `Int#`*)* [Peyton Jones and Launchbury 1991]. The mechanism for enforcing the restriction was the *kind system*: `Int` has kind $\star$ (the kind of ordinary data types), but `Int#` has a different kind # (the kind of unboxed data). GHC then required that the kind of a quantified type variable t could be $\star$ or, say, $(\star \to \star)$, *but never* #.

Since this is the same problem as the one we encountered for arity, in function *poly* in Section 3.1, we might expect the same solution to work. And indeed that is the approach adopted by Downen et al. [2019]: type variables cannot range over arrow types $\tau \rightsquigarrow \sigma$, whose kind is different from $\star$.

While this approach has served GHC well for nearly three decades, it has two major inadequacies:

(1) Too restrictive. Consider the *error* function, which prints a message and prematurely ends the program. It can have the type $\forall t.\mathtt{String} \to t$. Because *error* never returns a value, it doesn't matter how t is represented; truly the same code can be used no matter if t is boxed or unboxed. But the draconian restriction on unboxed types rejects this kind of polymorphism. Instead, specialized versions are needed, like *errorInt*`#` : `String` $\to$ `Int#`, even though the compiled code is identical. For *error* we would prefer to be able to say that the representation of t doesn't matter.

(2) Kind polymorphism. In a language with kind polymorphism, we can write a term with type $\forall k.\forall(a : k).\tau$. Here, a can have *any* kind k, including #. We have now lost our ability to prevent quantifiers over #-types, and need a new solution to the problem of uncompilable polymorphism.[3]

[3] A particularly knowledgeable reader might be aware that GHC supported kind polymorphism and restricted quantification for a few years. This worked because a type like $\forall k(a{:}k).a \to a$ is ill-kinded; the $\to$ type former puts requirements on a. However, with *kind equalities* [Weirich et al. 2013], a type variable $a : k$ can be cast to a more specific kind, causing chaos. In fact, this interaction originally spurred the development of Eisenberg and Peyton Jones [2017].

3.4 A Better Way: Look to the Kinds

Fortunately, both (1) and (2) can be solved, by using…more polymorphism. Let's start with representation polymorphism, using the approach of Eisenberg and Peyton Jones [2017].[4] We give `Int` and `Int#` these more refined kinds:

$$\texttt{Int} : \texttt{TYPE PtrR} \qquad \texttt{Int\#} : \texttt{TYPE IntR}$$

where $\texttt{TYPE} : \texttt{Rep} \to \star$, and $\texttt{PtrR}, \texttt{IntR} : \texttt{Rep}$.[5] The idea is that, given $\tau : \texttt{TYPE}\ r$, values of type τ have a runtime representation described by r. Now we can quantify over types represented by a heap pointer with $\forall(a : \texttt{TYPE PtrR}).\tau$. But we can *also* define representation-polymorphic functions—like $error : \forall(r : \texttt{Rep})(a : \texttt{TYPE}\ r).\ \texttt{String} \to a$—solving problem (1).

What of problem (2)? Instead of limiting how type variables can be instantiated (which is incompatible with full kind polymorphism), we instead add side conditions to the typing rules for abstraction and application, excluding uncompilable programs like so:

$$\frac{\Gamma, x{:}\tau \vdash e : \sigma \quad \boxed{\Gamma \vdash \tau\ \textbf{mono-rep}}}{\Gamma \vdash \lambda x{:}\tau.e : \tau \rightsquigarrow \sigma}\ \textsc{Fun-I} \qquad \frac{\Gamma \vdash e : \tau \rightsquigarrow \sigma \quad \Gamma \vdash e' : \tau \quad \boxed{\Gamma \vdash \tau\ \textbf{mono-rep}}}{\Gamma \vdash e\ e' : \sigma}\ \textsc{Fun-E}$$

The τ **mono-rep** caveat says τ's kind must be representation-monomorphic; that is, it can be `TYPE PtrR`; or `TYPE IntR`; but *not* `TYPE` r. For example, we might try to give *binapp* this type

$$binapp : \forall(r_a, r_b, r_c : \texttt{Rep})(a : \texttt{TYPE}\ r_a)(b : \texttt{TYPE}\ r_b)(c : \texttt{TYPE}\ r_c).(a \to b \to c) \to a \to b \to c$$
$$binapp\ f\ x\ y = f\ x\ y$$

But that should be rejected because we cannot compile the call $(f\ x\ y)$ without knowing how x and y are represented (and thus, how to retrieve them and pass them to f). Indeed, it is rejected (by *both* Fun-I and Fun-E). On the other hand, perhaps surprisingly, this type for *binapp* is fine:

$$binapp : \forall(r_c : \texttt{Rep})(a : \texttt{TYPE PtrR})(b : \texttt{TYPE PtrR})(c : \texttt{TYPE}\ r_c).(a \to b \to c) \to a \to b \to c$$

Notice that *binapp* can be representation-polymorphic in the return type c. Because our compiler supports tail-call elimination, f is the one to return a value of type c to the caller, not *binapp*.

3.5 Arity Polymorphism

Arity and representations share problems (1) and (2), and thankfully, they share solutions, too. We add a second parameter to `TYPE` that describes the arity of the function, like this:[6]

$$poly : \forall(a : \texttt{TYPE PtrR Call[2]}).\ (\texttt{Int} \rightsquigarrow \texttt{Int} \rightsquigarrow a) \rightsquigarrow (a, a)$$
$$poly = \Lambda(a : \texttt{TYPE PtrR Call[2]}).\lambda(f : \texttt{Int} \rightsquigarrow \texttt{Int} \rightsquigarrow a).\ \textbf{let}\ g : \texttt{Int} \rightsquigarrow a = f\ 3\ \textbf{in}\ (g\ 5, g\ 4)$$

Here, a's kind says that it only ranges over arity-2 types, of kind `TYPE PtrR Call[2]`, while f's type $\texttt{Int} \rightsquigarrow \texttt{Int} \rightsquigarrow a$ has kind `TYPE PtrR Call[4]`, hence f has arity 4 in total. Similarly, $g : \texttt{Int} \rightsquigarrow a$ of kind `TYPE PtrR Call[3]` has arity 3. In short, *the calling convention of a function (how many arguments to pass simultaneously) is described by its kind.* Of course, this meant that we had to choose a particular arity for a, just as we had to choose a particular representation for the a and b in *binapp*'s type. We enforce this "particular arity" constraint not at the point of abstraction, but at the point of application. Here is the augmented Fun-E rule:

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow \sigma \quad \Gamma \vdash e' : \tau \quad \Gamma \vdash \tau\ \textbf{mono-rep} \quad \boxed{\Gamma \vdash \tau\ \textbf{mono-conv}}}{\Gamma \vdash e\ e' : \sigma}\ \textsc{Fun-E}$$

where the new side condition τ **mono-conv** ensures `TYPE`'s second parameter is statically known. Preparing an argument of function type is precisely the point at which a compiler must compile

[4]The paper *Levity Polymorphism* in fact describes *representation polymorphism*. For levity polymorphism, see Section 3.6.
[5]GHC classifies representations like `PtrR`, `IntR`, *etc.*, as `RuntimeRep`; here, we shorten the name to just `Rep`.
[6]These kinds are somewhat simplified; the full story is in Section 4.

code for that argument that actually takes the number of arguments specified by its kind. Requiring the calling convention be monomorphic fixes that number statically; it can't be a type variable, say.

3.6 Evaluation Strategy and Levity

We have discussed *arity* and *representation*, two of our three design axes. But what of *levity*?

When compiling a function call $f\ (1+1)$, we must know the evaluation strategy to use: do we evaluate $(1+1)$ *before* calling f or *after*, on-demand? Programming languages typically commit to a choice here, which is usually to evaluate $(1+1)$ before the function call—the eager, call-by-value strategy. Haskell makes the opposite choice, implementing the lazy, call-by-need strategy; $(1+1)$ is evaluated only when its value is needed, and only once. But regardless of a language's choice of evaluation strategy, some scenarios require the opposite. Programmers in eager languages sometimes use constructs like *Delay* and *Force* [Wadler et al. 1998] to embed lazy evaluation, and programmers in lazy languages manually introduce strictness (such as Haskell's *seq*) to force eager evaluation. So a compiler's intermediate language should support both lazy and eager evaluation, and ideally without favoring either over the other.

We can use types (or, more precisely, their kinds) to control evaluation order. For example, suppose we have types $\mathtt{Int}^{\mathsf{L}}$ and $\mathtt{Int}^{\mathsf{U}}$, where the L stands for "Lifted" (the type has an extra bottom element, $\bot$, denoting divergent computation) and U for "Unlifted" (the type has no extra bottom element). Operationally, a value of lifted type must be represented as a pointer to a heap-allocated object because it may be an unevaluated thunk; a value of unlifted type may well still be represented by a pointer, but to the value itself, not a thunk. Variables of unlifted types cannot be bound to (divergent) computations, so they must be evaluated eagerly at binding-time; variables of lifted types may be bound to any unevaluated computation, so those bindings may be evaluated lazily. The types of an eager language (like OCaml) are all unlifted, whereas the types of a lazy language (like Haskell) are all lifted. But our $\mathcal{IL}$ supports both, and hence can be a target for both.

Once again, we can track levity in the kinds, like this:

$$\mathtt{Int}^{\mathsf{L}} : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathtt{Eval}^{\mathsf{L}} \qquad \mathtt{Int}^{\mathsf{U}} : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathtt{Eval}^{\mathsf{U}} \qquad \mathtt{Int\#} : \mathsf{TYPE}\ \mathsf{IntR}\ \mathtt{Eval}^{\mathsf{U}}$$

The levity of a type is all about its evaluation strategy. We already know primitive functions cannot be evaluated without calling them (Section 2.4), and thus a type has *either* a levity *or* an arity. We accordingly re-use the second component of the kind TYPE, which in Section 3.5 described the arity of a primitive function, using $\mathtt{Eval}^{\mathsf{L}}$ and $\mathtt{Eval}^{\mathsf{U}}$ to classify lifted and unlifted data types respectively. Reflecting this dual use, we describe the second argument of TYPE as the kind's *convention*, connoting "calling convention" for functions and "evaluation convention" for data.

Once again, we can be polymorphic in both levity (Section 4.4) and convention (Section 4.5). And we still keep the program compilable despite type erasure using suitable side conditions (Section 4.6). In the case of levity, the **mono-conv** premise in Fun-E (recall Section 3.5) is exactly what is needed to specify whether to compile this application using call-by-need or call-by-value.

3.7 From $\mathcal{IL}$ to $\mathcal{ML}$

Returning to arities, how can *poly* in Section 3.5 be compiled? In particular, since g's kind specifies that it has arity 3, we must compile g to code that takes three arguments. So, before generating code, we η-expand g. But apparently we can't, at least not in the confines of $\mathcal{IL}$'s type system, because a is not an arrow type!

To have any hope of solving this problem, we must do more than say a is an arity-2 type. We must also spell out the representations of its two arguments, so that we can generate code for passing the η-expanded function arguments. In *poly*, the explicated calling convention looks like:

$$\begin{aligned} poly = \Lambda(a : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathtt{Call}[\mathsf{PtrR}, \mathsf{IntR}]).\ \lambda(f : \mathtt{Int} \rightsquigarrow \mathtt{Int} \rightsquigarrow a). \\ \textbf{let}\, g : \mathtt{Int} \rightsquigarrow a = f\ 3\ \ \textbf{in}\ (g\ 5, g\ 4) \end{aligned}$$

$$
\begin{array}{rlrl}
\mathit{Kind} \ni & \kappa ::= \texttt{TYPE}\ \rho\ \nu & & \\
\mathit{Type} \ni & \tau, \sigma ::= t \mid T_p \mid T_d \mid \tau \rightsquigarrow \sigma \mid {}^{\gamma}\{\tau\} \mid \forall \chi.\sigma & \mathit{TyVar} \ni & \chi ::= t{:}\kappa \mid r \mid g \mid n \\
\mathit{Representation} \ni & \rho ::= r \mid \texttt{PtrR} \mid \texttt{IntR} \mid \ldots & \mathit{PrimType} \ni & T_p ::= \texttt{Int\#} \mid \ldots \\
\mathit{Levity} \ni & \gamma ::= g \mid \texttt{L} \mid \texttt{U} & \mathit{DataType} \ni & T_d ::= \texttt{Int}^{\gamma} \mid \ldots \\
\mathit{Convention} \ni & \nu ::= n \mid \texttt{Eval}^{\gamma} \mid \texttt{Call}[\alpha] & \mathit{Arity} \ni & \alpha ::= \rho, \alpha \mid \varepsilon \mid \mathit{arity}(\nu)
\end{array}
$$

$$
\begin{array}{rl}
\mathit{Expr} \ni & e ::= x \mid c \mid \texttt{I\#}^{\gamma}\ e \mid \textbf{case}\ e\ \textbf{of}\ \texttt{I\#}\ x \to e' \mid \lambda x{:}\tau.e \mid e\ e' \mid \texttt{Clos}^{\gamma}\ e \mid \texttt{App}\ e \mid \Lambda\chi.e \mid e\ \phi \\
\mathit{Const} \ni & c ::= i \mid op \qquad\qquad \mathit{PrimOp} \ni op ::= \texttt{error} \mid \ldots
\end{array}
$$

$$
\mathit{Answer} \ni A ::= x \mid c \mid \texttt{I\#}^{\gamma}\ A \mid \texttt{Clos}^{\gamma}\ e \mid \Lambda\chi.A \mid A\ \phi \qquad \mathit{Erasable} \ni \phi ::= \tau \mid \rho \mid \gamma \mid \nu
$$

Fig. 1. Syntax of $\mathcal{IL}$: An intermediate language with levity, representation, and arity polymorphism

The $\texttt{Call}[\texttt{PtrR}, \texttt{IntR}]$ in a's kind describes the representation of its two arguments.

Before code generation, we compile from $\mathcal{IL}$ into a lower-level representation $\mathcal{ML}$ (suggesting "machine language"). $\mathcal{ML}$ is still statically typed, but much more coarsely: $\mathcal{ML}$'s types correspond to representations and calling conventions. So compiled polymorphic code might look like:

$$
\begin{aligned}
\mathit{poly} = \lambda(f{:}\texttt{PtrR}).\ &\textbf{let}\ g : \texttt{PtrR} = \lambda(x{:}\texttt{PtrR},\ y{:}\texttt{PtrR},\ z{:}\texttt{IntR}).\ f(3, x, y, z) \\
&\textbf{in}\ (\lambda(y{:}\texttt{PtrR},\ z{:}\texttt{IntR}).\ g(5, y, z),\quad \lambda(y{:}\texttt{PtrR},\ z{:}\texttt{IntR}).\ g(4, y, z))
\end{aligned}
$$

In $\mathcal{ML}$ the functions are uncurried, and specify all their arguments and their representations. As such, every function is fully η-expanded, and every call is fully saturated. This representation is less convenient for the optimizer (for which we use $\mathcal{IL}$), but is just right for the code generator. We discuss $\mathcal{ML}$, and the lowering transformation from $\mathcal{IL}$ to $\mathcal{ML}$, in Section 6.

4 THE INTERMEDIATE LANGUAGE ($\mathcal{IL}$)

Our intermediate language, which we call $\mathcal{IL}$, is an explicitly-typed λ-calculus based closely on System F, with its syntax (Fig. 1), formation of types (Fig. 2), and type system (Fig. 3). We cover the kind system of $\mathcal{IL}$ in more detail in Sections 4.2 to 4.7.

The syntax for expressions e is given in Fig. 1, and includes the following forms.

- Variables x, and constants c.
- *Primitive integers* have type $\texttt{Int\#} : \texttt{TYPE IntR Eval}^{\texttt{U}}$, and are represented by a machine integer (hence $\texttt{IntR}$ in the kind). There are numeric constants, $i : \texttt{Int\#}$, and primitive functions (op) that operate over $\texttt{Int\#}$, such as $\texttt{plus\#} : \texttt{Int\#} \rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#}$.
- *Boxed integers* have type $\texttt{Int}^{\gamma} : \texttt{TYPE PtrR Eval}^{\gamma}$, where the levity γ can be $\texttt{L}$, $\texttt{U}$ or a levity variable g. A data-constructor application $\texttt{I\#}^{\gamma}\ e : \texttt{Int}^{\gamma}$ allocates a box containing the value of e (see Section 3.6). Such $\texttt{I\#}$ boxes are unpacked by a pattern-matching **case**.
- *Primitive functions* are introduced and eliminated with the familiar forms $\lambda x{:}\tau.e$ and $e\ e'$. As discussed in Section 2.2 we use a wavy arrow ($\tau \rightsquigarrow \sigma$) to remind ourselves that in this λ-calculus functions have some unusual behavior: they enjoy unrestricted η-expansion. Multiple arguments, even though they are curried, are always passed simultaneously; *e.g.,* if $f : \texttt{Int\#} \rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#}$, then f is always called with all three arguments.
- *Function closures* are introduced and eliminated with the forms $\texttt{Clos}^{\gamma}\ e$ and $\texttt{App}\ e$, respectively. Similar to boxed integers, $\texttt{Clos}^{\gamma}\ e$ allocates a new closure containing e, where γ records the levity of the computation responsible for that allocation, just as with $\texttt{I\#}^{\gamma}$.
- *Type abstraction* $\Lambda\chi.e$ *and application* $e\ \phi$, have the same meaning as in System F. While their operational interpretation in $\mathcal{IL}$ is analogous to $\lambda x{:}\tau.e$ and $e\ e'$, these abstractions and applications are fully *erased* by compilation to a lower-level representation. The only new

Formation rules for kinds, levities, representations, conventions, arities:

$$\boxed{\Gamma \vdash \kappa\ \mathbf{kind}} \qquad \frac{\Gamma \vdash \rho\ \mathbf{rep} \quad \Gamma \vdash \nu\ \mathbf{conv}}{\Gamma \vdash \mathsf{TYPE}\ \rho\ \nu\ \mathbf{kind}}$$

$$\boxed{\Gamma \vdash \rho\ \mathbf{rep}} \qquad \frac{}{\Gamma, r \vdash r\ \mathbf{rep}} \qquad \frac{}{\Gamma \vdash \mathtt{PtrR}\ \mathbf{rep}} \qquad \frac{}{\Gamma \vdash \mathtt{IntR}\ \mathbf{rep}}$$

$$\boxed{\Gamma \vdash \gamma\ \mathbf{lev}} \qquad \frac{}{\Gamma, g \vdash g\ \mathbf{lev}} \qquad \frac{}{\Gamma \vdash \mathsf{L}\ \mathbf{lev}} \qquad \frac{}{\Gamma \vdash \mathsf{U}\ \mathbf{lev}}$$

$$\boxed{\Gamma \vdash \nu\ \mathbf{conv}} \qquad \frac{}{\Gamma, n \vdash n\ \mathbf{conv}} \qquad \frac{\Gamma \vdash \gamma\ \mathbf{lev}}{\Gamma \vdash \mathtt{Eval}^{\gamma}\ \mathbf{conv}} \qquad \frac{\Gamma \vdash \alpha\ \mathbf{ari}}{\Gamma \vdash \mathtt{Call}[\alpha]\ \mathbf{conv}}$$

$$\boxed{\Gamma \vdash \alpha\ \mathbf{ari}} \qquad \frac{\Gamma \vdash \rho\ \mathbf{rep} \quad \Gamma \vdash \alpha\ \mathbf{ari}}{\Gamma \vdash \rho, \alpha\ \mathbf{ari}} \qquad \frac{}{\Gamma \vdash \varepsilon\ \mathbf{ari}} \qquad \frac{\Gamma \vdash \nu\ \mathbf{conv}}{\Gamma \vdash \mathit{arity}(\nu)\ \mathbf{ari}}$$

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

$$\frac{\Gamma \vdash \kappa\ \mathbf{kind}}{\Gamma, t{:}\kappa \vdash t : \kappa}\ \textsc{TVar} \qquad \frac{}{\Gamma \vdash \mathtt{Int\#} : \mathsf{TYPE}\ \mathtt{IntR}\ \mathtt{Eval}^{\mathsf{U}}}\ \textsc{Int\#} \qquad \frac{\Gamma \vdash \gamma\ \mathbf{lev}}{\Gamma \vdash \mathtt{Int}^{\gamma} : \mathsf{TYPE}\ \mathtt{PtrR}\ \mathtt{Eval}^{\gamma}}\ \textsc{Int}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{TYPE}\ \rho_1\ \nu_1 \quad \Gamma \vdash \tau_2 : \mathsf{TYPE}\ \rho_2\ \nu_2}{\Gamma \vdash \tau_1 \rightsquigarrow \tau_2 : \mathsf{TYPE}\ \mathtt{PtrR}\ \mathtt{Call}[\rho_1, \mathit{arity}(\nu_2)]}\ \textsc{Fun} \qquad \frac{\Gamma \vdash \gamma\ \mathbf{lev} \quad \Gamma \vdash \tau : \mathsf{TYPE}\ \rho\ \nu}{\Gamma \vdash {}^{\gamma}\{\tau\} : \mathsf{TYPE}\ \mathtt{PtrR}\ \mathtt{Eval}^{\gamma}}\ \textsc{Clo}$$

$$\frac{\Gamma, \chi \vdash \sigma : \mathsf{TYPE}\ \rho\ \nu \quad \Gamma \vdash \mathsf{TYPE}\ \rho\ \nu\ \mathbf{kind}}{\Gamma \vdash \forall \chi.\sigma : \mathsf{TYPE}\ \rho\ \nu}\ \textsc{Forall} \qquad \frac{\Gamma \vdash \tau : \kappa \quad \kappa = \kappa'}{\Gamma \vdash \tau : \kappa'}\ \textsc{K-Conv}$$

Reflexivity, transitivity, symmetry, and compatibility for $\boxed{\kappa = \kappa'}$, plus the following rules:

$$\mathit{arity}(\mathtt{Eval}^{\gamma}) = \varepsilon \qquad \mathit{arity}(\mathtt{Call}[\alpha]) = \alpha$$

Monomorphism restrictions:

$$\frac{\Gamma \vdash \tau : \mathsf{TYPE}\ \rho\ \nu \quad \vdash \rho\ \mathbf{rep}}{\Gamma \vdash \tau\ \mathbf{mono\text{-}rep}}\ \textsc{Mono-Rep} \qquad \frac{\Gamma \vdash \tau : \mathsf{TYPE}\ \rho\ \nu \quad \vdash \nu\ \mathbf{conv}}{\Gamma \vdash \tau\ \mathbf{mono\text{-}conv}}\ \textsc{Mono-Conv}$$

Fig. 2. Kind and levity system of $\mathcal{IL}$

addition here is that binders χ and arguments ϕ range over four different sorts of type-level variables and arguments (all erasable). The expressiveness and restrictions on these different kinds of polymorphism are detailed in Sections 4.3 to 4.8. Type binders are annotated with their kinds, and other binders are distinguished by their naming convention; we use a Latin name for type-level variables and the corresponding Greek name for the syntactic category. For clarity, we sometimes state the sort of variable being quantified (r:`Rep`, g:`Lev`, or n:`Conv`).

A subset of these expressions are *Answers* (denoted by A, Fig. 1), meaning that they are possible results of evaluation. This classification accounts for the eventual *type erasure* mentioned above. The apparent redex $(\Lambda\chi.A)\ \phi$ is an answer, because it is erased to A at compile-time. Answers include *closures* $\mathtt{Clos}^{\gamma}\ e$ as usual, but not primitive functions $\lambda x{:}\tau.e$; recall that functions are *called* instead of *evaluated*, and thus cannot be answers of evaluation (Section 2.4).

The rules for well-formed types and kinds are given in Fig. 2, and include type variables t, primitive types T_p (of which we supply one, `Int#`), algebraic data types T_d (of which we supply one, `Int`), polymorphic types $\forall\chi.\sigma$, primitive function types $\tau \rightsquigarrow \sigma$ and a closure type ${}^{\gamma}\{\tau\}$.

$\mathcal{IL}$'s type system given in Fig. 3 ensures the following guarantees for well-behaved programs:

- *Static Compilation*: Every well-typed program can be compiled to a lower-level representation for a monomorphic machine, which models details such as specialized registers (for integers versus pointers) and function calls with multiple arguments (Theorem 3).
- *Type Safety*: If a well-typed program is equal to a number (as per Section 4.9), then its compiled code computes that number (Theorem 4). In particular, executing well-typed programs never

Types of expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma, x{:}\tau \vdash x : \tau}\ \textsc{Var} \qquad \frac{c : \tau}{\Gamma \vdash c : \tau}\ \textsc{Const}$$

$$\frac{\Gamma \vdash e : \texttt{Int\#}}{\Gamma \vdash \texttt{I\#}^{\gamma}\, e : \texttt{Int}^{\gamma}}\ \textsc{Int-I} \qquad \frac{\Gamma \vdash e : \texttt{Int}^{\gamma} \quad \Gamma, x{:}\texttt{Int\#} \vdash e' : \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \texttt{I\#}\ x \to e' : \tau}\ \textsc{Int-E}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \sigma \quad \Gamma \vdash \tau\ \textbf{mono-rep}}{\Gamma \vdash \lambda x{:}\tau.e : \tau \rightsquigarrow \sigma}\ \textsc{Fun-I} \qquad \frac{\Gamma \vdash e : \tau \rightsquigarrow \sigma \quad \Gamma \vdash A : \tau \quad \Gamma \vdash \tau\ \textbf{mono-rep}}{\Gamma \vdash e\ A : \sigma}\ \textsc{Fun-A-E}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow \sigma \quad \Gamma \vdash e' : \tau \quad \Gamma \vdash \tau\ \textbf{mono-rep} \quad \Gamma \vdash \tau\ \textbf{mono-conv}}{\Gamma \vdash e\ e' : \sigma}\ \textsc{Fun-E}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau\ \textbf{mono-conv}}{\Gamma \vdash \texttt{Clos}^{\gamma}\ e : {}^{\gamma}\{\tau\}}\ \textsc{Clo-I} \qquad \frac{\Gamma \vdash e : {}^{\gamma}\{\tau\}}{\Gamma \vdash \texttt{App}\ e : \tau}\ \textsc{Clo-E}$$

$$\frac{\Gamma, \chi \vdash e : \sigma \quad \Gamma \vdash \forall\chi.\sigma : \kappa}{\Gamma \vdash \Lambda\chi.e : \forall\chi.\sigma}\ \forall\text{-I} \qquad \frac{\Gamma \vdash e : \forall\chi.\sigma \quad \Gamma \vdash [\phi/\chi]\ \textbf{poly}}{\Gamma \vdash e\ \phi : \sigma[\phi/\chi]}\ \forall\text{-E}$$

Polymorphic instantiations: $\boxed{\Gamma \vdash [\phi/\chi]\ \textbf{poly}}$

$$\frac{\Gamma \vdash \gamma\ \textbf{lev}}{\Gamma \vdash [\gamma/g]\ \textbf{poly}} \qquad \frac{\Gamma \vdash \rho\ \textbf{rep}}{\Gamma \vdash [\rho/r]\ \textbf{poly}} \qquad \frac{\Gamma \vdash \nu\ \textbf{conv}}{\Gamma \vdash [\nu/n]\ \textbf{poly}} \qquad \frac{\Gamma \vdash \tau : \kappa \quad \Gamma \vdash \kappa\ \textbf{kind}}{\Gamma \vdash [\tau/t{:}\kappa]\ \textbf{poly}}$$

Types of constants: $\boxed{c : \tau}$

$$i : \texttt{Int\#} \qquad \texttt{error} : \forall r.\forall g.\forall(t : \texttt{TYPE}\ r\ \texttt{Eval}^{g}).\ \texttt{Int\#} \rightsquigarrow t$$

Fig. 3. Type system of $\mathcal{IL}$

gets stuck, because the correct kind of register is used to store each value, and primitive functions are always called with the correct number and kind of arguments.

Static compilation requires some typing rules in Fig. 3 to explicitly refer to the *kinds of types*, as described in Fig. 2, that can be assigned to certain expressions. This shows up in the occasional τ **mono-rep** and τ **mono-conv** side conditions, which we elaborate in Section 4.6. Intuitively, τ **mono-rep** captures the fact that τ has a *statically-known, monomorphic representation*, and τ **mono-conv** says that τ has a *statically-known, monomorphic convention.*

4.1 Simplifying Assumptions

To maintain a minimal presentation of $\mathcal{IL}$, we make many simplifying assumptions that reduce its number of features to a small representative core illustrating our main objectives. A realistic implementation will include more features, which can either be added as an extension to $\mathcal{IL}$, or encoded in terms of the features shown here. Our simplifying assumptions are as follows:

- Higher kinds (*e.g.,* $\star \to \star$) are not included, but are a standard extension orthogonal to $\mathcal{IL}$.
- `Int#` is the only exemplary primitive unboxed type, which introduces the only non-pointer representation `IntR`. Other atomic unboxed types and representations—say, for floating-point numbers, characters, arrays, *etc.*—can be added straightforwardly, as can primitive operations on these types, like `plus#` and `minus#` of type $\texttt{Int\#} \rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#}$. Unboxed tuple types $(\#\tau_1, \ldots, \tau_n\#)$ are an extension that introduces a compound representation roughly dual to the calling convention of functions: $\texttt{Call}[\rho_1, \ldots, \rho_n]$ describes the representations of values that a function needs to consume, while $\texttt{Tuple}[\rho_1, \ldots, \rho_n]$ [Eisenberg and Peyton Jones 2017] describes the representations of values that a tuple has within it.
- $\texttt{Int}^{\gamma}$ serves as the only example of an algebraic data type, which happens to be parameterized by a levity γ specifying whether the result of the constructor is evaluated eagerly (`U`) or

lazily (L). In general, there should be a way to declare new user-defined algebraic data types. These need not be levity polymorphic (*i.e.,* have a γ parameter), but $\mathcal{IL}$ makes it possible to combine levity polymorphism with user-defined data types; see Section 4.8.

- There is no built-in **let**-binding construct but, as usual, a non-recursive **let** can be regarded as shorthand for λ and application: $\mathbf{let}\, x{:}\tau = e \,\mathbf{in}\, e' \triangleq (\lambda x{:}\tau.\, e')\ e$. The typing rules for **let**-bindings can be derived from the rules for primitive function types $\tau \rightsquigarrow \sigma$. As such, **let**-bindings inherit similar side conditions on the type of the bound variable; see Section 4.6.
- `error` is the only source of computational effects in $\mathcal{IL}$ as presented here. Recursive let-bindings, which are essential for practical functional programming, can be added straightforwardly, as can other computational effects, such as printing and state as in OCaml, with additional primitive operations.
- Because `error` is the only side-effect in $\mathcal{IL}$, call-by-name and call-by-need evaluation always give the same result [Ariola and Felleisen 1997]. So in $\mathcal{IL}$, we interpret "lifted" (L) as call-by-need for operational concerns, and as call-by-name for equational reasoning. To accommodate richer side-effects, the choice should instead be made explicit. This can be done directly in $\mathcal{IL}$ by further dividing L into separate levities for call-by-name (CBN) and call-by-need (Need) evaluation. For example, evaluating $\mathbf{let}\, x{:}\mathtt{Int}^{\gamma} = (\mathit{print}\ \texttt{"bye "}; \mathtt{I\#}^{\gamma}\ 0)\ \mathbf{in}\ (\mathit{print}\ \texttt{"hi "}; x; x)$ prints: `"bye hi "` with $\gamma = \mathsf{U}$, `"hi bye bye "` with $\gamma = \mathsf{CBN}$, and `"hi bye "` with $\gamma = \mathsf{Need}$.

4.2 Kinds, Representations, Levities, and Conventions

A kind κ has the form TYPE ρ ν, where ρ describes the *representation* of the type, and ν describes its *convention*, that is, what operations are allowed on that type. Suppose $x : \tau : \mathsf{TYPE}\ \rho\ \nu$; that is, x is a term variable of type τ, whose kind is TYPE ρ ν. The representation ρ specifies the runtime representation of the value of x. Referring to Fig. 1, ρ can be:

- `PtrR`, meaning that x is a pointer into the garbage-collected heap.
- `IntR`, meaning that x is a machine integer (not a pointer).
- r, a representation variable bound by a $\forall$; that is, we support representation polymorphism [Eisenberg and Peyton Jones 2017].

The convention ν describes the allowed operations on x, *i.e.,* how it can be consumed. ν can be:

- $\mathtt{Eval}^{\mathsf{U}}$, meaning that x cannot be bound to a computation like $\bot$ (hence U for "Unlifted"). This kind is used for primitive values, and heap pointers that point directly to the value itself (such as a heap-allocated array).
- $\mathtt{Eval}^{\mathsf{L}}$, meaning that x may be bound to a computation like $\bot$ (hence L for "Lifted"). This kind is used for thunks, which might need evaluation to get its value, and might diverge doing so.
- $\mathtt{Eval}^{g}$, where g is a levity variable bound by $\forall$; that is, we support levity polymorphism.
- $\mathtt{Call}[\alpha]$, meaning that x is a primitive function (not a thunk) with an *arity* described by α. The arity of an $\mathcal{IL}$ primitive function might be a fixed non-empty list $\rho_1, \ldots, \rho_m$, so x takes precisely $m \geq 1$ arguments,[7] with representations given by ρ_i. Otherwise, the arity is $\rho_1, \ldots, \rho_m, \mathit{arity}(\nu)$, meaning that x takes *at least m* arguments with the listed representations, followed by possibly some more arguments given by the arity of ν.
- n, a convention variable bound by $\forall$; that is, we support convention polymorphism.

The kind TYPE PtrR $\mathtt{Eval}^{\mathsf{L}}$ expresses the uniform representation of a value in a lazy language: a pointer to a lifted (*i.e.,* possibly a thunk) object stored in the heap. Because this kind is so common,

[7]There is no type in $\mathcal{IL}$ with the convention `Call[]`, but it can easily arise in extensions of $\mathcal{IL}$. In practice, unboxed tuple arguments are passed simultaneously in several registers. So the type (# `Bool`, `Int#`, `String` #) $\rightsquigarrow$ $\mathtt{Int}^{\mathsf{L}}$ can be given the kind TYPE PtrR `Call[PtrR, IntR, PtrR]`. The nullary case of unboxed tuple arguments, (# #) $\rightsquigarrow$ $\mathtt{Int}^{\mathsf{L}}$, can then be given the kind TYPE PtrR `Call[]`.

we often abbreviate it to $\star$ for the default kind. In a call-by-value language we would instead define the default kind $\star$ as $\mathtt{TYPE\ PtrR\ Eval}^{\mathsf{U}}$, and $\mathtt{Eval}^{\mathsf{L}}$ would be used sparingly, if at all.

4.3 Calling Conventions in Kinds

We track the arity of a function type in its kind, as described in Sections 3.5 and 3.7. For example:

$$\begin{array}{rcl} \mathtt{Int\#} \rightsquigarrow \mathtt{Int\#} & : & \mathtt{TYPE\ PtrR\ Call[IntR]} \\ \mathtt{Int\#} \rightsquigarrow \mathtt{Int\#} \rightsquigarrow \mathtt{Int\#} & : & \mathtt{TYPE\ PtrR\ Call[IntR, IntR]} \\ \mathtt{Int\#} \rightsquigarrow \mathtt{Int}^{\mathsf{L}} \rightsquigarrow \mathtt{Int\#} & : & \mathtt{TYPE\ PtrR\ Call[IntR, PtrR]} \end{array}$$

The convention of each of these primitive function types has the form $\mathtt{Call}[\alpha]$, where the arity given by α describes the arguments needed to fully call functions of that type.

The formation rule for $(\tau_1 \rightsquigarrow \tau_2)$ keeps track of these arities. Looking at rule FUN in Fig. 2, we see the kind of $(\tau_1 \rightsquigarrow \tau_2)$ has a calling convention of $\mathtt{Call}[\rho_1, \mathit{arity}(\nu_2)]$. Two special cases are:

$$\frac{\Gamma \vdash \tau_1 : \mathtt{TYPE}\ \rho_1\ \nu \quad \Gamma \vdash \tau_2 : \mathtt{TYPE}\ \rho'\ \mathtt{Call}[\rho_2, \ldots, \rho_m]}{\Gamma \vdash \tau_1 \rightsquigarrow \tau_2 : \mathtt{TYPE\ PtrR\ Call}[\rho_1, \rho_2, \ldots, \rho_m]} \qquad \frac{\Gamma \vdash \tau_1 : \mathtt{TYPE}\ \rho_1\ \nu \quad \Gamma \vdash \tau_2 : \mathtt{TYPE}\ \rho'\ \mathtt{Eval}^{\gamma}}{\Gamma \vdash \tau_1 \rightsquigarrow \tau_2 : \mathtt{TYPE\ PtrR\ Call}[\rho_1]}$$

The representation of the first argument (ρ_1) is that of τ_1. The rest of the arguments come from the convention ν_2 of τ_2, via the type-level operation *arity*, as defined in Fig. 2. It returns the arity α of $\mathtt{Call}[\alpha]$; and the empty arity in the case of $\mathtt{Eval}^{\gamma}$. But ν_2 might also be a variable n, and then $\mathit{arity}(n)$ is stuck; that is why $\mathit{arity}(\nu)$ is part of the syntax of α in Fig. 1. Rule K-CONV allows calls to *arity* to be calculated whenever desired.[8]

4.4 Polymorphism in Levity and Representation

We are used to polymorphism over types, but we can gainfully employ polymorphism over levities, representations, and conventions, which is extremely useful in practice. For example, levity polymorphism lets us write some functions that work uniformly over both strict and lazy values. Adding two boxed integers can be defined as

$$\mathit{plus} : \forall g_1. \forall g_2. \forall g_3. \mathtt{Int}^{g_1} \rightsquigarrow \mathtt{Int}^{g_2} \rightsquigarrow \mathtt{Int}^{g_3} \qquad \mathit{plus}\ (\mathtt{I\#}\ x)\ (\mathtt{I\#}\ y) = \mathtt{I\#}\ (\mathtt{plus\#}\ x\ y)$$

which is short-hand for the following definition in $\mathcal{IL}$:

$$\begin{array}{l} \mathit{plus}\ :\ \forall g_1. \forall g_2. \forall g_3. \mathtt{Int}^{g_1} \rightsquigarrow \mathtt{Int}^{g_2} \rightsquigarrow \mathtt{Int}^{g_3} \\ \mathit{plus} = \Lambda g_1. \Lambda g_2. \Lambda g_3. \lambda(x' : \mathtt{Int}^{g_1}). \lambda(y' : \mathtt{Int}^{g_2}).\ \mathbf{case}\ x'\ \mathbf{of}\ \mathtt{I\#}\ x \to \mathbf{case}\ y'\ \mathbf{of}\ \mathtt{I\#}\ y \to \mathtt{I\#}^{g_3}\ (\mathtt{plus\#}\ x\ y) \end{array}$$

Notice that in this definition, the levities g_i of the argument and return types are statically unknown, so we must be able to pattern-match on and return values with unknown levities. Specifically, rule INT-E in Fig. 3 allows a case-expression to scrutinize an integer of arbitrary levity γ. Operationally, a levity-polymorphic **case** has to test the scrutinee to see if it is a thunk (in case the levity variable is instantiated to L), and if so evaluate it. In essence, we can interpret a **case** on an unknown levity as a lifted one because a **case** is always strict and, if it happens that g is U, the branch for handling a thunk is simply dead code.[9]

Similarly, suppose we had a primitive type of arrays, $\mathtt{Array\#}^{\gamma}$, with kinding rule

$$\frac{\Gamma \vdash \gamma\ \mathbf{lev} \quad \Gamma \vdash \tau : \mathtt{TYPE\ PtrR}\ \nu}{\Gamma \vdash \mathtt{Array\#}^{\gamma}\ \tau : \mathtt{TYPE\ PtrR\ Eval}^{\gamma}}$$

[8] Alternatively, we could require $\mathit{arity}(\nu)$ to be fully calculated in the FUN kinding rule. This would let us remove $\mathit{arity}(\nu)$ from the grammar of arities, but also forces an additional restriction on the formation of types and expressions, specifically FUN and FUN-I, to rule out $\mathit{arity}(n)$. Such a restriction comes with the cost of breaking a pleasant property of $\mathcal{IL}$: except for the $\forall$ quantifier, any type made from well-kinded types is itself well-kinded.

[9] We assume that the concrete, runtime representation of values (not thunks) is the same for both eager and lazy evaluation. This is true in GHC and seems likely in other systems supporting laziness, but it may not hold in some systems.

From a representation point of view, an $\texttt{Array\#}^{\gamma}$ is represented by a pointer and contains pointers. The array itself can be lifted or unlifted, and (independently) can contain lifted or unlifted values. For example, the type $\texttt{Array\#}^{\mathsf{L}}\ (\texttt{Array\#}^{\mathsf{U}}\ \texttt{Int}^{\mathsf{U}})$ is a lifted array of pointers, each of which points directly to an array of pointers to (boxed) integers. The ability to exclude the possibility of intermediate thunks in this data structure is very valuable in high-performance code, as a recent spate of GHC proposals shows [Eisenberg 2019; Graf 2020; Martin 2019a,b,c; Theriault 2019].

4.5 Polymorphism in Convention

We may also be polymorphic in conventions. Consider the reverse-apply function

$$\mathit{revapp}\ x\ f = f\ x$$

For now, suppose that it returns `Int#`. What type should *revapp* have? Here are two possibilities:

$$\begin{array}{ll}(1) & \mathit{revapp} : \forall (t : \texttt{TYPE PtrR Eval}^{\mathsf{L}}).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#} \\ (2) & \mathit{revapp} : \forall (t : \texttt{TYPE IntR Eval}^{\mathsf{U}}).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#}\end{array}$$

We want to compile a function like *revapp* to a single block of efficient machine code. To do so, *we must know the representation of x*, because we have to generate instructions to move x around. If x is represented by an integer, it will be passed in one sort of register; if a float, in another; if a pointer then yet another.[10] So we can choose (1) or (2), but not both.

On the other hand, consider these other possible types:

$$\begin{array}{ll}(3) & \mathit{revapp} : \forall (t : \texttt{TYPE PtrR Eval}^{\mathsf{U}}).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#} \\ (4) & \mathit{revapp} : \forall (t : \texttt{TYPE PtrR Call[IntR]}).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#}\end{array}$$

Since we are simply moving x around, but not otherwise acting upon it, we *can* simultaneously allow (1), (3), and (4). That is, we can be completely polymorphic in its convention n, thus:

$$(4) \quad \mathit{revapp} : \forall (n : \texttt{Conv})\ (t : \texttt{TYPE PtrR}\ n).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#}$$

What about the return type of f? The code for *revapp* does not manipulate f's return value at all (it does not even move it around, thanks to tail-call elimination), so it can be completely polymorphic in the representation or t_2, giving this type:

$$(5) \quad \mathit{revapp} : \forall (n{:}\texttt{Conv})\ (r{:}\texttt{Rep})\ (g{:}\texttt{Lev})\ (t_1{:}\texttt{TYPE PtrR}\ n)\ (t_2{:}\texttt{TYPE}\ r\ \texttt{Eval}^{g}).\ t_1 \rightsquigarrow (t_1 \rightsquigarrow t_2) \rightsquigarrow t_2$$

But notice that, unlike the argument type t_1, *revapp* cannot be polymorphic in the convention of t_2, because a compiler needs to statically know the arity of the λ-bound argument f to generate code for $(f\ x)$ inside *revapp* (Section 4.6). It can, however, be evaluated with any levity.

4.6 Restrictions on Polymorphism

Of course we have the usual restrictions on polymorphism,[11] but $\mathcal{IL}$'s polymorphism introduces some new issues. We have already seen how unrestricted polymorphism is incompatible with efficient static code generation[12] in Section 4.5, where we cannot allow *revapp*'s type argument t_1 to have a representation-polymorphic kind. A second restriction is demonstrated by:

$$\mathit{twice}\ f\ x = f\ (f\ x)$$

Should $(f\ x)$ be eagerly or lazily evaluated? If it has a lifted type, then we can build a thunk for it, and pass that thunk to f. Otherwise, we must evaluate it before the call—remember, variables of

[10] Even if pointers occupy the same sort of register as integers, they are treated quite differently by the garbage collector, so the code generator treats them differently.

[11] *E.g.*, every free variable (of every sort) that appears anywhere in the type checking judgment $\Gamma \vdash e : \tau$ must be bound by Γ.

[12] Runtime code generation would allow the system to clone fresh code for each representationally-distinct instantiation of a function. But this is a pretty big hammer: only .NET does this. To keep things simple, we assume static code generation.

unlifted types are always bound to values, never thunks. So we can give *twice* either of these types:

$$\begin{array}{ll}(1) & \mathit{twice} : \forall(t{:}\mathsf{TYPE\ PtrR\ Eval}^{\mathsf{L}}).\ (t \leadsto t) \leadsto t \leadsto t \\ (2) & \mathit{twice} : \forall(t{:}\mathsf{TYPE\ PtrR\ Eval}^{\mathsf{U}}).\ (t \leadsto t) \leadsto t \leadsto t\end{array}$$

but we must choose: unlike *revapp*, *twice* cannot be polymorphic in t's convention.

The restrictions on polymorphism used to ensure static compilability are embodied in the shaded premises in the type system of Fig. 3. The judgment $\Gamma \vdash \tau$ **mono-rep**, defined in Fig. 2 checks that the representation of τ is monomorphic; that is, that it mentions no variables. This is ensured by the empty context in the second premise of rule MONO-REP. There is an equivalent judgment $\Gamma \vdash \tau$ **mono-conv** for conventions. Now returning to Fig. 3 we see the shaded premises:

- Rule FUN-E: for a general application, the argument type must be monomorphic in both the representation (so that we know how to store it while passing), and convention (so that we know when to evaluate it, *or* for first-class primitive function arguments, how to create it).
- Rule FUN-A-E: in the special case where the argument syntactically has the form of an answer, we can allow the argument type to be levity-polymorphic, since in this case there is no distinction between lazy or eager. If the application is lazy, then the argument does not need to be evaluated anyway. If application is eager, then the argument is already a value (or a variable that must be bound to a value), and again, does not need to be evaluated.
- Rule CLO-I: this rule boxes a primitive function so, just as in FUN-E, we must know that function's arity statically, so we can compile code for it that starts by taking the correct number of arguments.

We can justify these restrictions intuitively, but how do we know that these are the "right" restrictions? To answer that question we will show, in Section 6, how to compile $\mathcal{IL}$ into our lower level $\mathcal{ML}$. In the translation from $\mathcal{IL}$ to $\mathcal{ML}$ in Fig. 8, we need exactly the shaded monomorphic restrictions of Fig. 3. If any of these restrictions were removed, then there would be expressions that are well-typed, yet uncompilable.

Our rules also include two additional, unshaded, monomorphism restrictions, in the FUN-I and FUN-A-E rules. These restrictions enforce an extra invariant on the environment Γ: *every variable in* Γ *has a monomorphic representation.* Besides making intuitive sense, this invariant could be necessary in a compiler accounting for more low-level details like storing free variables in a closure; doing so certainly requires knowing their representation. However, perhaps shockingly, the compilation scheme we give in Section 6 *does not* require any monomorphism restrictions in FUN-I and FUN-A-E: they could be deleted and yet all closed, well-typed expressions could still be compiled. This example suggests different compilers might need different restrictions on polymorphism. And from the reverse standpoint, other compilation schemes might allow for new and more adventurous possibilities for levity, representation, and convention polymorphism.

As an example of why different restrictions are needed for the FUN-E and FUN-A-E rules, recall the encoding of **let**-bindings from Section 4.1, which gives us the typing rules

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash e' : \sigma \quad \boxed{\Gamma \vdash \tau\ \textbf{mono-rep}} \quad \boxed{\Gamma \vdash \tau\ \textbf{mono-conv}}}{\Gamma \vdash \textbf{let}\ x{:}\tau = e\ \textbf{in}\ e' : \sigma}\ \textsc{Let} \qquad \frac{\Gamma \vdash A : \tau \quad \Gamma, x{:}\tau \vdash e : \sigma \quad \boxed{\Gamma \vdash \tau\ \textbf{mono-rep}}}{\Gamma \vdash \textbf{let}\ x{:}\tau = A\ \textbf{in}\ e : \sigma}\ \textsc{Let-A}$$

derived by composing FUN-I with FUN-E or FUN-A-E, respectively. That is, in general a **let** can bind a variables to expressions with both a monomorphic representation and convention; but convention-polymorphic answers can also be bound. Neither typing rule is more general than the other: LET-A only applies to some bound expressions (pre-evaluated answers), whereas LET only applies to bindings of some types (ones with statically-known conventions).

Now, consider these **let**-bindings, which follow the restrictions of either Let or Let-A above:

(1) $\textbf{let}\, x : \forall g.\mathtt{Int}^g$
$\quad x = \Lambda g.\ \mathtt{I\#}^g\, 0\ \textbf{ in }\ \ldots$

(2) $\textbf{let}\, x : \mathtt{Int}^{\mathsf{L}}$
$\quad x = \mathit{plus}\,\mathsf{L}\,\mathsf{L}\,\mathsf{L}\ (\mathtt{I\#}^{\mathsf{L}}\, 1)\ (\mathtt{I\#}^{\mathsf{L}}\, 2)\ \textbf{ in }\ \ldots$

(3) $\textbf{let}\, x : \forall n.\forall(t : \mathsf{TYPE}\ \mathsf{PtrR}\ n).t$
$\quad x = y\ \textbf{ in }\ \ldots$

(4) $\textbf{let}\, x : \forall g.\mathtt{Int}^g \rightsquigarrow \mathtt{Int}^g$
$\quad x = \Lambda g.\lambda(y : \mathtt{Int}^g).\ f\ (f'\ y)\ \textbf{ in }\ \ldots$

Both (1) and (3) bind x to syntactically manifest answers with convention-polymorphic types ($\forall g.\mathtt{Int}^g : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Eval}^g$ and $\forall n.\forall(t{:}\mathsf{TYPE}\ \mathsf{PtrR}\ n).\ t : \mathsf{TYPE}\ \mathsf{PtrR}\ n$, respectively), which is well-typed by Let-A. In contrast, (2) and (3) bind x to non-answer expressions, so x needs a convention-monomorphic type (here, $\mathtt{Int}^{\mathsf{L}} : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Eval}^{\mathsf{L}}$ and $\forall g.\ \mathtt{Int}^g \rightsquigarrow \mathtt{Int}^g : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Call}[\mathsf{PtrR}]$, respectively) in order to be well-typed by the Let rule.

4.7 The Forall Rule

The polymorphic quantifier $\forall t{:}\kappa.\sigma$ has no impact on a type's kind: it just inherits the kind of σ (rule Forall in Fig. 2). Intuitively, this is because these quantifiers will be totally *erased* by compilation, and have no impact on the final runtime code; the $\forall$ is "invisible" to the lower-level machine.

However, now that variables may appear in kinds, we must be careful they do not escape their scope. For example, the following type is not well-kinded:

$$\forall r.\ \forall(t{:}\mathsf{TYPE}\ r\ \mathsf{Eval}^{\mathsf{L}}).\ t \rightsquigarrow t \quad :? \quad \mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Call}[r]$$

Here the representation r of the first parameter escapes in the calling convention of this primitive function type, because r is meant to be local to the type itself. This nonsense is prevented by the second premise of rule Forall in Fig. 2 and the second premise of rule $\forall$-I in Fig. 3.

4.8 User-Defined Types and Code Reuse

As we have seen, polymorphism over these type descriptors—representation, convention, and levity—increases the opportunities for code reuse. We can even have *data types* that are polymorphic over these descriptors, although this is beyond the scope of the $\mathcal{IL}$ we formally describe here. For example definition of boxed integers (given in Section 3.2) might be generalized to

$$\begin{array}{l}\textbf{data}\ \mathtt{Int}\ (g : \mathsf{Lev}) : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Eval}^g\ \textbf{where}\\ \quad \mathtt{I\#} : \mathtt{Int\#} \rightsquigarrow \mathtt{Int}\ g\end{array}$$

Now, the $\mathtt{Int}$ type is parameterized by a chosen levity ($g : \mathsf{Lev}$), which determines whether or not the boxed integers are evaluated eagerly or lazily.

Polymorphism over a constructor's arguments and result can be combined within a single definition. For example, here is a further generalized definition of lists whose spine can be either eagerly or lazily evaluated (g), and containing elements of any arity or evaluation strategy (n):

$$\begin{array}{l}\textbf{data}\ \mathtt{List}\ (g : \mathsf{Lev})\ (n : \mathsf{Conv})\ (t : \mathsf{TYPE}\ \mathsf{PtrR}\ n) : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Eval}^g\ \textbf{where}\\ \quad \mathtt{Nil} : \mathtt{List}\ g\ n\ t\\ \quad \mathtt{Cons} : t \rightsquigarrow \mathtt{List}\ g\ n\ t \rightsquigarrow \mathtt{List}\ g\ n\ t\end{array}$$

Despite restrictions on polymorphism, we can define some levity-polymorphic functions over this type. For example, we could write the following polymorphic definition which is capable of summing up a list of integers with any combination of levities:

$$\begin{array}{lcl}\mathit{sum} & : & \forall g_1\ g_2\ g_3.\ \mathtt{List}\ g_1\ \mathsf{Eval}^{g_2}\ (\mathtt{Int}\ g_2) \rightsquigarrow \mathtt{Int}\ g_3\\ \mathit{sum}\ \mathtt{Nil} & = & \mathtt{I\#}\ 0\\ \mathit{sum}\ (\mathtt{Cons}\ (\mathtt{I\#}\ x)\ xs) & = & \textbf{case}\ \mathit{sum}\ xs\ \textbf{of}\ \mathtt{I\#}\ y \longrightarrow \mathtt{I\#}\ (\mathtt{plus\#}\ x\ y)\end{array}$$

sum's caller chooses the levity of the list's spine (g_1), the list's elements (g_2), and the result (g_3). This is possible because *sum* is completely strict anyway; if it is given an evaluated list or an unevaluated

Type-based definition of substitutable values: $\boxed{\Gamma \vdash e\ \mathbf{subst}}$

$$\frac{}{\Gamma \vdash A\ \mathbf{subst}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \mathtt{TYPE}\ \rho\ \mathtt{Eval}^{\mathtt{L}}}{\Gamma \vdash e\ \mathbf{subst}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \mathtt{TYPE}\ \rho\ \mathtt{Call}[\alpha]}{\Gamma \vdash e\ \mathbf{subst}}$$

Equational axioms (in each rule, assume that $\Gamma \vdash S$ **subst**):

$$\begin{array}{llll}
(\beta_{\rightsquigarrow}) & (\lambda x{:}\tau.e)\ S = e[S/x] & (\eta_{\rightsquigarrow}) & \lambda x{:}\tau.(e\ x) = e : \tau \rightsquigarrow \sigma \\
(\beta_{\forall}) & (\Lambda\chi.e)\ \phi = e[\phi/\chi] & (\eta_{\forall}) & \Lambda\chi.(e\ \chi) = e : \forall\chi.\sigma \\
(\beta_{\{\}}) & \mathtt{App}\ (\mathtt{Clos}^{\gamma}\ e) = e & (\eta_{\{\}}) & \mathtt{Clos}^{\gamma}\ (\mathtt{App}\ S) = S : {}^{\gamma}\{\sigma\} \\
(\beta_{\mathtt{Int}}) & \begin{array}{l}\mathbf{case}\ \mathtt{I\#}^{\gamma}\ A\ \mathbf{of} \\ \quad \mathtt{I\#}\ x \to e\end{array} = e[A/x] & (\eta_{\mathtt{Int}}) & \begin{array}{l}\mathbf{case}\ e\ \mathbf{of} \\ \quad \mathtt{I\#}\ x \to \mathtt{I\#}^{\gamma}\ x\end{array} = e : \mathtt{Int}^{\gamma}
\end{array}$$

Plus closure under reflexivity, transitivity, symmetry, and compatibility.

Fig. 4. Equational theory of $\mathcal{IL}$

thunk, the entire thing will be added together before a value is returned. Therefore, the same code can be used for any combination of eager or lazy evaluation.

Other functions, such as the standard definition for mapping over a list:

$$\mathit{map}\ f\ \mathtt{Nil} = \mathtt{Nil} \qquad\qquad \mathit{map}\ f\ (\mathtt{Cons}\ x\ xs) = \mathtt{Cons}\ (f\ x)\ (\mathit{map}\ f\ xs)$$

cannot be so levity polymorphic. That's because the order of evaluation in $\mathtt{Cons}\ (f\ x)\ (\mathit{map}\ f\ xs)$ depends on the levity of $(f\ x)$ and the recursive call $(\mathit{map}\ f\ xs)$. We can give this type assignment to *map* that specifies its result should be lazy, like in Haskell:

$$\mathit{map} : \forall g\ n\ (a{:}\mathtt{TYPE}\ \mathtt{PtrR}\ n)\ (b{:}\mathtt{TYPE}\ \mathtt{PtrR}\ \mathtt{Eval}^{\mathtt{L}}).\ (a \rightsquigarrow b) \rightsquigarrow \mathtt{List}\ g\ n\ a \rightsquigarrow \mathtt{List}\ \mathtt{L}\ \mathtt{Eval}^{\mathtt{L}}\ b$$

Note that the input list can store values of any convention at all, and like with *sum*, it can be either spine-strict or spine-lazy; *map* is strict in its second argument either way. Other evaluation orders for *map* can be specified by replacing one or both of $\mathtt{L}$ in $(\mathtt{List}\ \mathtt{L}\ \mathtt{Eval}^{\mathtt{L}}\ b)$ with $\mathtt{U}$. Although the definition of *map* appears the same in $\mathcal{IL}$, this change will compile to very different machine code.

4.9 Equational Theory

The equational theory for $\mathcal{IL}$, as defined in Fig. 4, gives us a framework to reason about equivalent $\mathcal{IL}$ expressions. We use this as the basis for correctness of compiling a high-level language to $\mathcal{IL}$ (Theorem 2) and further on to a low-level language (Theorem 3). The rules for function closures (namely $\beta_{\{\}}$ and $\eta_{\{\}}$) and boxed integers ($\beta_{\mathtt{Int}}$ and $\eta_{\mathtt{Int}}$) are unsurprising, as are the rules for erasable abstractions ($\beta_{\forall}$ and $\eta_{\forall}$). More distinctive is $\eta_{\rightsquigarrow}$, which (as discussed in Section 2) allows unrestricted η in either direction for *any* expression of a primitive function type $\tau \rightsquigarrow \sigma$.

That leaves the reduction rule $\beta_{\rightsquigarrow}$. As is usual in a call-by-value λ-calculus, only some expressions—called *substitutable values*, denoted by the metavariable S—can be passed to a primitive function and substituted for its formal parameter by the $\beta_{\rightsquigarrow}$ rule. The $\beta_{\rightsquigarrow}$ rule only fires if the argument of the application is substitutable. Unlike most systems—using a purely syntactic definition of substitutable values—$\mathcal{IL}$ identifies these expressions by their *type* and *kind*, as defined by the rules for the $\Gamma \vdash e$ **subst** judgment. Using the type and kind of the argument to define substitutability, rather than only its syntax, allows us to integrate several different evaluation orders (call-by-value, call-by-name, *etc.*) within the same language.

Consider the rules for the $\Gamma \vdash e$ **subst** judgment. Firstly, we designate all *answers* A to be *substitutable*. Notice that, up to type erasure, each answer is considered a value in the call-by-value λ-calculus. In other cases, the criteria for substitutability of a typed term $e : \tau$ depends on the convention of τ rather than the syntax of e. For example, in a call-by-name setting, every expression can be substituted for a variable. So in $\mathcal{IL}$, *all* lazily-evaluated arguments—which have the convention $\mathtt{Eval}^{\mathsf{L}}$—are substitutable values, and hence allow $\beta_{\rightsquigarrow}$ to fire. In contrast, the only values in call-by-value languages are answers (A), so variables of type $\mathtt{Eval}^{\mathsf{U}}$ can only be substituted with answers. More generally, answers are always substitutable in all of the evaluation strategies we are interested in here, so we can say something more: A is substitutable for *all* conventions. This extra step is helpful in case we are dealing with an expression—like x or $\mathtt{I\#}^{g}\ i$—which has an unknown convention but will inevitably be substitutable in any case.

For example, consider the different evaluation orders of a function call with lifted or unlifted arguments. On the one hand, we can express a lazy call to *plus* (Section 4.4) such as

$$(\lambda x{:}\mathtt{Int}^{\mathsf{L}}.e)\ (\mathit{plus}\,\mathsf{L}\ (\mathtt{I\#}^{\mathsf{L}}\,1)\ (\mathtt{I\#}^{\mathsf{L}}\,2)) =_{\beta_{\rightsquigarrow}} e[\mathit{plus}\,\mathsf{L}\ (\mathtt{I\#}^{\mathsf{L}}\,1)\ (\mathtt{I\#}^{\mathsf{L}}\,2)/x]$$

which substitutes the unevaluated argument for the parameter x right away according to $\beta_{\rightsquigarrow}$. This is possible because the argument has the type $\mathtt{Int}^{\mathsf{L}} : \mathsf{TYPE}\ \mathtt{PtrR}\ \mathtt{Eval}^{\mathsf{L}}$, and so it is substitutable by virtue of its type. On the other hand, the corresponding eager call would be

$$(\lambda x{:}\mathtt{Int}^{\mathsf{U}}.e)\ (\mathit{plus}\,\mathsf{U}\ (\mathtt{I\#}^{\mathsf{U}}\,1)\ (\mathtt{I\#}^{\mathsf{U}}\,2)) = (\lambda x{:}\mathtt{Int}^{\mathsf{U}}.e)\ (\mathtt{I\#}^{\mathsf{U}}\,3) =_{\beta_{\rightsquigarrow}} e[\mathtt{I\#}^{\mathsf{U}}\,3/x]$$

Here, we cannot apply the $\beta_{\rightsquigarrow}$ directly as before, because the argument is not substitutable: it has the type $\mathtt{Int}^{\mathsf{U}} : \mathsf{TYPE}\ \mathtt{PtrR}\ \mathtt{Eval}^{\mathsf{U}}$ but is not an answer. Instead we must first evaluate the argument; the result is the answer $(\mathtt{I\#}^{\mathsf{U}}\,3)$, and $\beta_{\rightsquigarrow}$ can now fire.

5 COMPILATION TO $\mathcal{IL}$ FROM A HIGHER LEVEL

$\mathcal{IL}$ is, by design, a fairly low-level language making fine distinctions about representation, calling conventions, evaluation orders, and so on. This makes it a target for *both* eager *and* lazy languages. To see how, we now give translations for call-by-name and call-by-value System F into $\mathcal{IL}$.

5.1 Call-by-Name System F to $\mathcal{IL}$

To translate call-by-name System F into $\mathcal{IL}$, we begin by picking a single "uniform" $\mathcal{IL}$ kind $\star$ that captures all the types of the source language, namely $\star = \mathsf{TYPE}\ \mathtt{PtrR}\ \mathtt{Eval}^{\mathsf{L}}$. Each source-language type τ translates to an $\mathcal{IL}$ type $[\![\tau]\!]$ of kind $\star$; that is, a pointer to a lifted value, perhaps a thunk.

Fig. 5 gives this type translation. To get the correct call-by-name semantics for numbers, we use the boxed integer type $\mathtt{Int}^{\mathsf{L}}$, which happily has the correct kind. However, even though the primitive function type $[\![\tau]\!] \rightsquigarrow [\![\sigma]\!]$ has the correct call-by-name semantics, it has the wrong kind $\mathsf{TYPE}\ \mathtt{PtrR}\ \mathtt{Call[PtrR]}$, so the translation coerces $\mathtt{Call[PtrR]}$ to $\mathtt{Eval}^{\mathsf{L}}$ with a closure type. Polymorphic type abstraction is unchanged, only clarified that bound type variable ranges over $\star$.

To compile expressions, we only need to expand out the additions prescribed by the translation of types. Numeric constants need to be boxed, functions and their calls need the explicit coercions to and from closures, and bound type variables are annotated with their uniform kind.

5.2 Call-by-Value System F to $\mathcal{IL}$

We can compile a call-by-value version of system F using virtually the same procedure as above. Again, we need to decide on a uniform kind that is suitable for each source-level type, which is $\star = \mathsf{TYPE}\ \mathtt{PtrR}\ \mathtt{Eval}^{\mathsf{U}}$ for call-by-value evaluation. As before, we can compile source-level types following the invariant that $[\![\tau]\!] : \star$, as again shown in Fig. 5.

Note that we still compile integers to a boxed type, so that all values are represented uniformly by a pointer, but this time we make it unlifted to reflect the call-by-value semantics. Function types are still wrapped in a closure, as in the call-by-name case, but this time unlifted ones.

The translation of polymorphism is more complex, however, due a mismatch with the semantics of call-by-value System F. With call-by-value evaluation, the abstraction $\Lambda t.\bot$ is a value, even though $\bot$ diverges, whereas in call-by-name they would be η-equivalent. We need to make sure

Call-by-name where $\star$ = $\mathtt{TYPE\ PtrR\ Eval}^{\mathsf{L}}$

$$\begin{array}{llll}
[\![\mathtt{Int}]\!] \triangleq \mathtt{Int}^{\mathsf{L}} \quad [\![t]\!] \triangleq t & [\![x]\!] \triangleq x & [\![i]\!] \triangleq \mathtt{I\#}^{\mathsf{L}}\ i \\
[\![\tau \to \sigma]\!] \triangleq {}^{\mathsf{L}}\{[\![\tau]\!] \rightsquigarrow [\![\sigma]\!]\} & [\![\lambda x{:}\tau.e]\!] \triangleq \mathtt{Clos}^{\mathsf{L}}\,(\lambda x{:}[\![\tau]\!].[\![e]\!]) & [\![e\ e']\!] \triangleq (\mathsf{App}\ [\![e]\!])\ [\![e']\!] \\
[\![\forall t.\tau]\!] \triangleq \forall t{:}\star.[\![\tau]\!] & [\![\Lambda t.e]\!] \triangleq \Lambda t{:}\star.[\![e]\!] & [\![e\ \tau]\!] \triangleq [\![e]\!]\ [\![\tau]\!]
\end{array}$$

Call-by-value where $\star$ = $\mathtt{TYPE\ PtrR\ Eval}^{\mathsf{U}}$

$$\begin{array}{llll}
[\![\mathtt{Int}]\!] \triangleq \mathtt{Int}^{\mathsf{U}} \quad [\![t]\!] \triangleq t & [\![x]\!] \triangleq x & [\![i]\!] \triangleq \mathtt{I\#}^{\mathsf{U}}\ i \\
[\![\tau \to \sigma]\!] \triangleq {}^{\mathsf{U}}\{[\![\tau]\!] \rightsquigarrow [\![\sigma]\!]\} & [\![\lambda x{:}\tau.e]\!] \triangleq \mathtt{Clos}^{\mathsf{U}}\,(\lambda x{:}[\![\tau]\!].[\![e]\!]) & [\![e\ e']\!] \triangleq (\mathsf{App}\ [\![e]\!])\ [\![e']\!] \\
[\![\forall t.\tau]\!] \triangleq {}^{\mathsf{U}}\{\forall t{:}\star.[\![\tau]\!]\} & [\![\Lambda t.e]\!] \triangleq \mathtt{Clos}^{\mathsf{U}}\,(\Lambda t{:}\star.[\![e]\!]) & [\![e\ \tau]\!] \triangleq (\mathsf{App}\ [\![e]\!])\ [\![\tau]\!]
\end{array}$$

Fig. 5. Compiling call-by-name and call-by-value System F to $\mathcal{IL}$

Call-by-name substitutable values: Call-by-value substitutable values:

$$V ::= e \qquad\qquad V ::= x \mid \lambda x{:}\tau.e \mid \Lambda t.e$$

Equational axioms (for both call-by-name and -value definitions of V):

$$\begin{array}{llll}
(\beta_{\to}) & (\lambda x{:}\tau.e)\ V = e[V/x] & (\eta_{\to}) & \lambda x{:}\tau.(V\ x) = V : \tau \to \sigma \\
(\beta_{\forall}) & (\Lambda t.e)\ \tau = e[\tau/t] & (\eta_{\forall}) & \Lambda t.(V\ t) = V : \forall t.\sigma \\
& (name) & (\lambda x{:}\tau.e\ x)\ e' = e\ e' &
\end{array}$$

Fig. 6. Equational theory of System F; call-by-value and call-by-name

that this abstraction is still a value even after the polymorphic Λ is erased. For that reason, we must introduce the additional call-by-value closure which is preserved to runtime.

The compilation of call-by-value expressions is nearly the same as call-by-name expressions. Besides swapping L for U, the only difference is in for polymorphic abstractions and instantiations. These have an extra closure that signifies call-by-value evaluation and, more importantly, makes sure that the value $\Lambda t.e$ in System F compiles to a value, namely $\mathtt{Clos}^{\mathsf{U}}\,(\Lambda t{:}\star.[\![e]\!])$, which is essential when side effects or non-termination enters the picture. For example, evaluating $\Lambda t.\mathtt{error}\ t\ i$ returns immediately in call-by-value System F, but the corresponding $\Lambda t{:}\star.\mathtt{error}\ \mathtt{PtrR}\ \mathtt{U}\ t\ i$ causes an error in $\mathcal{IL}$. Intuitively, the explicit closure and application serve to codify the standard definition of type erasure for call-by-value System F, which traditionally erases $\Lambda t.e$ into $\lambda().e$.

5.3 Correctness of Source-to-$\mathcal{IL}$ Compilation

Compiling call-by-name and call-by-value System F into $\mathcal{IL}$ are both correct: type checking and equalities are preserved by translation. We assume the standard type system for System F and semantics in Fig. 6. Note that the *name* axiom, which gives a name to the argument of a function, corresponds to a right-to-left evaluation order for the call-by-value semantics, and is a consequence of $\beta_{\to}$ in the call-by-name semantics. The preservation of types is straightforward, where the compilation of a typing environment $[\![\Gamma]\!]$ is defined pointwise.

Theorem 1 (Type Preservation)**.** *If* $\Gamma \vdash e : \tau$ *is derivable then so is* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$.

Theorem 2 (Soundness and Completeness)**.** $\Gamma \vdash e = e' : \tau$ *in call-by-name System F if and only if* $[\![\Gamma]\!] \vdash [\![e]\!] = [\![e']\!] : [\![\tau]\!]$ *via call-by-name compilation, and likewise for call-by-value.*

6 COMPILATION FROM $\mathcal{IL}$ TO A LOWER LEVEL

In order to illustrate how our levity-polymorphic intermediate language $\mathcal{IL}$ might be compiled to a more conventional, lower-level language, we introduce an abstract machine that supports

$$
\begin{array}{l}
\mathit{PrimRep} \ni \pi ::= \mathtt{PtrR} \mid \mathtt{IntR} \quad \mathit{Strictness} \ni \psi ::= \mathtt{L} \mid \mathtt{U} \quad \mathit{KnownConv} \ni \eta ::= \mathtt{Eval}^{\psi} \mid \mathtt{Call}[\overline{\pi}] \\
\mathit{Expr} \ni e ::= W \mid W(\overline{a}) \mid \mathsf{App}\, e(\overline{a}) \mid \mathbf{case}\, e\, \mathbf{of}\, \mathtt{I\#}(x_{\mathtt{IntR}}) \to e' \mid \mathbf{let}\, x_{\pi}^{\psi} = e\, \mathbf{in}\, e' \\
\mathit{Reference} \ni R ::= \mathtt{I\#}(a) \mid \mathtt{Clos}^{n}\, W \mid \lambda(\overline{x_{\pi}}).e \\
\mathit{Value} \ni V ::= c \mid \mathtt{I\#}(a) \mid \mathtt{Clos}^{n}\, W \qquad \mathit{WHNF} \ni W ::= R \mid a \qquad \mathit{Arg} \ni a ::= c \mid x_{\pi} \\
\\
\mathit{StackCont} \ni K ::= \varepsilon \mid \mathsf{App}(\overline{a}); K \mid \mathbf{case}\, \mathtt{I\#}(x_{\mathtt{IntR}}) \to e; K \mid \mathbf{let}\, x_{\pi}\, \mathbf{in}\, e; K \mid \mathbf{set}\, x; K \\
\mathit{Store} \ni H ::= \varepsilon \mid [x := R]H \mid [x := \mathbf{memo}\, e]H \\
\mathit{MachineState} \ni m ::= \langle e \mid K \mid H \rangle \mid \mathtt{error}(n) \qquad \mathit{Final} \ni \mathit{Fin} ::= \langle V \mid \varepsilon \mid H \rangle \mid \mathtt{error}(n)
\end{array}
$$

$$
\begin{array}{lrcll}
(\mathit{PshApp}) & \langle \mathsf{App}\, e(\overline{a}) \mid K \mid H \rangle & \mapsto & \langle e \mid \mathsf{App}(\overline{a}); K \mid H \rangle & \\
(\mathit{PshCase}) & \langle \mathbf{case}\, e\, \mathbf{of}\, \mathtt{I\#}(x_{\mathtt{IntR}}) \to e' \mid K \mid H \rangle & \mapsto & \langle e \mid \mathbf{case}\, \mathtt{I\#}(x_{\mathtt{IntR}}) \to e'; K \mid H \rangle & \\
(\mathit{PshLet}) & \langle \mathbf{let}\, x_{\pi}^{\mathtt{U}} = e\, \mathbf{in}\, e' \mid K \mid H \rangle & \mapsto & \langle e \mid \mathbf{let}\, x_{\pi}\, \mathbf{in}\, e'; K \mid H \rangle & \\
(\mathit{LAlloc}) & \langle \mathbf{let}\, x_{\mathtt{PtrR}}^{\mathtt{L}} = e'\, \mathbf{in}\, e \mid K \mid H \rangle & \mapsto & \langle e[y_{\mathtt{PtrR}}/x_{\mathtt{PtrR}}] \mid K \mid [y := \mathbf{memo}\, e']H \rangle & \\
\\
(\mathit{Call}) & \langle (\lambda(\overline{x_{\pi}}).e)(\overline{a}) \mid K \mid H \rangle & \mapsto & \langle e[\overline{a/x_{\pi}}] \mid K \mid H \rangle & \\
(\mathit{Apply}) & \langle \mathtt{Clos}^{n}\, W \mid \mathsf{App}(\overline{a}); K \mid H \rangle & \mapsto & \langle W(\overline{a}) \mid K \mid H \rangle & (\text{if } n = |\overline{a}|) \\
(\mathit{Unbox}) & \langle \mathtt{I\#}(a) \mid \mathbf{case}\, \mathtt{I\#}(x_{\mathtt{IntR}}) \to e; K \mid H \rangle & \mapsto & \langle e[a/x_{\mathtt{IntR}}] \mid K \mid H \rangle & \\
(\mathit{Move}) & \langle c \mid \mathbf{let}\, x_{\pi}\, \mathbf{in}\, e; K \mid H \rangle & \mapsto & \langle e[c/x_{\pi}] \mid K \mid H \rangle & \\
(\mathit{SAlloc}) & \langle R \mid \mathbf{let}\, x_{\mathtt{PtrR}}\, \mathbf{in}\, e; K \mid H \rangle & \mapsto & \langle e[y_{\mathtt{PtrR}}/x_{\mathtt{PtrR}}] \mid K \mid [y := R]H \rangle & \\
\\
(\mathit{Fun}) & \langle y_{\mathtt{PtrR}}(\overline{a}) \mid K \mid [y := R]H \rangle & \mapsto & \langle R(\overline{a}) \mid K \mid [y := R]H \rangle & \\
(\mathit{Look}) & \langle y_{\mathtt{PtrR}} \mid K \mid [y := R]H \rangle & \mapsto & \langle R \mid K \mid [y := R]H \rangle & \\
(\mathit{Force}) & \langle y_{\mathtt{PtrR}} \mid K \mid [y := \mathbf{memo}\, e]H \rangle & \mapsto & \langle e \mid \mathbf{set}\, y; K \mid [y := \mathbf{memo}\, e]H \rangle & \\
(\mathit{Memo}) & \langle R \mid \mathbf{set}\, y; K \mid [y := \mathbf{memo}\, e]H \rangle & \mapsto & \langle R \mid K \mid [y := R]H \rangle & \\
\\
(\mathit{Error}) & \langle \mathtt{error}(n) \mid K \mid H \rangle & \mapsto & \mathtt{error}(n) &
\end{array}
$$

Fig. 7. Syntax and semantics for $\mathcal{ML}$

non-uniform representations and function arities, both of which must be monomorphic. The idea is to model a realistic machine architecture with multiple basic kinds of data representations (*e.g.,* pointers vs. integers), and where primitive functions are passed multiple arguments but are otherwise first order (though primitive function pointers may be passed as arguments and invoked). The syntax of this language, which we call $\mathcal{ML}$, is given in Fig. 7, along with its abstract machine.[13]

The syntax of $\mathcal{ML}$ is restricted from the more λ-calculus-inspired $\mathcal{IL}$ in several ways:

- Expressions follow the A-normal form (ANF) convention [Sabry and Felleisen 1993]: all arguments a are either variables or constants. To support ANF, $\mathcal{ML}$ has a **let** construct.
- Primitive functions ($\lambda(\overline{y_{\pi}}).e$) and calls ($W(\overline{a})$) have an explicit arity and pass multiple arguments at once, but cannot be partially applied.
- An applied function cannot be an arbitrary expression; it must be a weak head-normal form, namely a reference to a λ, variable, or constant. Every application $W(\overline{a})$ can be resolved in at most two steps: lookup W if it's a variable, then apply W if it's a λ or constant (like `error`).

As such, it is impossible to chain several calls in a row. For example, $f(1)(2)$ is not a legal expression in $\mathcal{ML}$. If f is an arity 2 primitive function, it must be called as $f(1,2)$; if it is an arity-1 primitive function returning a closure of an arity-1 primitive function, it must be called as $(\mathsf{App}\, f(1))(2)$.

[13]For aficionados of GHC, $\mathcal{IL}$ is like GHC's Core language, while $\mathcal{ML}$ is like the STG language [Peyton Jones 1992].

The number of arguments passed at once is explicitly fixed in each λ-abstraction and call site. In other words, $\mathcal{ML}$ does not support polymorphism of function arity.

$\mathcal{ML}$'s syntax includes annotations that make the semantically important information in types explicit, so the syntax of programs makes it clear how they are executed. In particular, each variable is annotated with its representation π, which must be either a pointer (PtrR) or integer (IntR), corresponding to an assignment to an appropriate register. In other words, all variables are permanently assigned a representation—intuitively, stored in either an integer or address register. By design, $\mathcal{ML}$ does not support polymorphism over these representations because the different types of machine registers are distinct, and the choice made is fixed in the code.

Additionally, each **let** binding is annotated as either eager (U) or lazy (L). This controls whether the right-hand side of the **let** is evaluated first before being bound to the variable, or bound first and evaluated later as needed. Again, this decision about evaluation order must be statically chosen for each **let**, so $\mathcal{ML}$ does not support polymorphism over evaluation order.

6.1 The Semantics of $\mathcal{ML}$

Executing an $\mathcal{ML}$ program involves a machine configuration of the form $\langle e \,|\, K \,|\, H \rangle$, where e is the expression being evaluated, K is the *continuation* or *call stack* of evaluation, and H is the heap for storing allocated memory. Heaps may contain both values ($[x := V]H$) or unevaluated thunks ($[x := \mathbf{memo}\, e]H$). Both call stacks and heaps are conventional, except that a stack may contain an application of many arguments in a single stack frame, like $\mathsf{App}(\overline{a}); K$. The other cases of stack frames include a **case**, a strict **let** binding, and a **set** construct to memoize thunk evaluation.

Many of the steps of the machine are also conventional, including those for pushing stack frames (*PshApp*, *PshCase*, *PshLet*) and allocating memory (*LAlloc*, *SAlloc*), but note that we include cases for *both* lazy bindings (*LAlloc*) and strict ones (*PshLet*, *SAlloc*). Next we have the rules for performing interesting reductions. *Apply* resolves the application of a closure by extracting the primitive function it contains, and *Call* calls a primitive function directly. Note that *Apply* can check that the number of arguments matches the arity of the closure at runtime (and potentially respond appropriately if they do not match, as we do later in Section 7). Instead, *Call* is merely undefined when the arguments don't match the bound parameters, representing a type or memory unsafe error. In addition, we have *Move* for moving a constant into an appropriate variable (corresponding to a register) and *Unbox* for extracting the contents of a boxed integer. Finally, we have the rules for handling pointer variables at runtime. *Fun* expects a function pointer to map to a value. For other pointers, we have to check if it is evaluated, to either *Look* up values or else *Force* thunks.[14] When a forced thunk returns a value, it is *Memo*ized to share the result on future uses.

6.2 Compilation

Compilation from $\mathcal{IL}$ to the low-level machine language $\mathcal{ML}$ is given in Fig. 8. The top-level translation is $\mathcal{E}_\nu[\![e]\!]^\Gamma_\theta$, which compiles a typed expression $\Gamma \vdash e : \tau$ given τ's convention is ν. The environment θ is a mapping from $\mathcal{IL}$ variables to $\mathcal{ML}$ arguments (either constants or representation-annotated variables) written as $[a_1/x_1] \dots [a_n/x_n]$.

A key part in understanding the compilation in Fig. 8 is to remember the distinction between calling and evaluating. In our system, only expressions with types like $\mathtt{Int\#}$, $\mathtt{Int}^\gamma$, and ${}^\gamma\{\tau\}$ can be evaluated. In contrast, expressions with types like $\tau \rightsquigarrow \sigma$ can only be called. Implementing this distinction is the main role of $\mathcal{E}_\nu[\![e]\!]^\Gamma_\theta$, which takes into account the convention ν of e: if it is $\mathtt{Eval}^\gamma$

[14]Note that this uniform check on pointers $y_{\mathtt{PtrR}}$ is needed to support levity polymorphism for types like $\mathtt{Int}^g$ and ${}^g\{\tau\}$. In a more practical compiler, we could have specialized code that avoids a check when it is statically known, due to type checking that $y_{\mathtt{PtrR}}$ must be unlifted, so that the *Look* step always applies without a dynamic check. Thus, a language which is call-by-value by default does not have to pay the runtime penalty for thunks unless they are actually being used.

In the following, all equations are tried left-to-right, top-to-bottom.

Top-level eta expansion:

$$\mathcal{E}_v[\![A]\!]^\Gamma_\theta \triangleq \mathcal{A}[\![A]\!]^\Gamma_\theta \qquad \mathcal{E}_{\mathtt{Eval}^\gamma}[\![e]\!]^\Gamma_\theta \triangleq \mathcal{C}[\![e]\!]^\Gamma_\theta(\varepsilon) \qquad \mathcal{E}_{\mathtt{Call}[\overline{\pi}]}[\![e]\!]^\Gamma_\theta \triangleq \lambda(\overline{x_\pi}).\mathcal{C}[\![e]\!]^\Gamma_\theta(\overline{x_\pi})$$

Constants and variables:

$$\mathcal{C}[\![i]\!]^\Gamma_\theta(\varepsilon) \triangleq i \qquad \mathcal{C}[\![x]\!]^{\Gamma,x:\tau}_\theta(\varepsilon) \triangleq \theta(x) \qquad (\text{if } \Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \mathtt{Eval}^\psi)$$

$$\mathcal{C}[\![\mathtt{error}]\!]^\Gamma_\theta(a) \triangleq \mathtt{error}(a) \quad \mathcal{C}[\![x]\!]^{\Gamma,x:\tau}_\theta(\overline{a}) \triangleq (\theta(x))(\overline{a}) \quad (\text{if } \Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \mathtt{Call}[\overline{\pi}])$$

Applications (the following equations are tried top-to-bottom):

$$\begin{aligned}
\mathcal{C}[\![\mathsf{App}\ e]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathsf{App}\ (\mathcal{C}[\![e]\!]^\Gamma_\theta(\varepsilon))(\overline{a}) \\
\mathcal{C}[\![e\ \phi]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathcal{C}[\![e]\!]^\Gamma_\theta(\overline{a}) \\
\mathcal{C}[\![e\ A]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathcal{C}[\![e]\!]^\Gamma_\theta(\mathcal{A}[\![A]\!]^\Gamma_\theta, \overline{a}) && (\text{if } \mathcal{A}[\![A]\!]^\Gamma_\theta = x_\pi \text{ or } \mathcal{A}[\![A]\!]^\Gamma_\theta = c) \\
\mathcal{C}[\![e\ A]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathbf{let}\, x^{\mathsf{U}}_{\mathtt{PtrR}} = \mathcal{A}[\![A]\!]^\Gamma_\theta \,\mathbf{in}\, \mathcal{C}[\![e]\!]^\Gamma_\theta(x_{\mathtt{PtrR}}, \overline{a}) \\
\mathcal{C}[\![e\ e']\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathbf{let}\, x^{lev(\eta)}_{\pi} = \mathcal{E}_\eta[\![e']\!]^\Gamma_\theta \,\mathbf{in}\, \mathcal{C}[\![e]\!]^\Gamma_\theta(x_\pi, \overline{a}) && (\text{if } \Gamma \vdash e':\tau,\ \Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi, \text{ and } \Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta)
\end{aligned}$$

Boxing and unboxing:

$$\mathcal{C}[\![\mathtt{I\#}^\gamma\ A]\!]^\Gamma_\theta(\varepsilon) \triangleq \mathcal{A}[\![\mathtt{I\#}^\gamma\ A]\!]^\Gamma_\theta$$

$$\mathcal{C}[\![\mathtt{I\#}^\gamma\ e]\!]^\Gamma_\theta(\varepsilon) \triangleq \mathbf{let}\, x^{\mathsf{U}}_{\mathtt{IntR}} = \mathcal{C}[\![e]\!]^\Gamma_\theta(\varepsilon) \,\mathbf{in}\, \mathtt{I\#}(x_{\mathtt{IntR}})$$

$$\mathcal{C}[\![\mathbf{case}\, e' \,\mathbf{of}\, \mathtt{I\#}\ x \to e]\!]^\Gamma_\theta(\overline{a}) \triangleq \mathbf{case}\, \mathcal{C}[\![e']\!]^\Gamma_\theta(\varepsilon) \,\mathbf{of}\, \mathtt{I\#}(x_{\mathtt{IntR}}) \to \mathcal{C}[\![e]\!]^{\Gamma,x:\mathtt{Int\#}}_{[x_{\mathtt{IntR}}/x]\theta}(\overline{a})$$

Abstractions:

$$\mathcal{C}[\![\lambda x{:}\sigma.e]\!]^\Gamma_\theta(a', \overline{a}) \triangleq \mathcal{C}[\![e]\!]^{\Gamma,x:\sigma}_{[a'/x]\theta}(\overline{a})$$

$$\mathcal{C}[\![\Lambda\chi.e]\!]^\Gamma_\theta(\overline{a}) \triangleq \mathcal{C}[\![e]\!]^{\Gamma,\chi}_\theta(\overline{a})$$

$$\mathcal{C}[\![\mathtt{Clos}^\gamma\ e]\!]^\Gamma_\theta(\varepsilon) \triangleq \mathcal{A}[\![\mathtt{Clos}^\gamma\ e]\!]^\Gamma_\theta$$

Type erasure of answers:

$$\mathcal{A}[\![c]\!]^\Gamma_\theta \triangleq c \qquad \mathcal{A}[\![x]\!]^\Gamma_\theta \triangleq \theta(x) \qquad \mathcal{A}[\![\mathtt{I\#}^\gamma\ A]\!]^\Gamma_\theta \triangleq \mathtt{I\#}(\mathcal{A}[\![A]\!]^\Gamma_\theta)$$

$$\mathcal{A}[\![A\ \phi]\!]^\Gamma_\theta \triangleq \mathcal{A}[\![A]\!]^\Gamma_\theta \qquad \mathcal{A}[\![\Lambda\chi.A]\!]^\Gamma_\theta \triangleq \mathcal{A}[\![A]\!]^\Gamma_\theta$$

$$\mathcal{A}[\![\mathtt{Clos}^\gamma\ e]\!]^\Gamma_\theta \triangleq \mathtt{Clos}^{|arity(\eta)|}\, \mathcal{E}_{\mathtt{Call}[arity(\eta)]}[\![e]\!]^\Gamma_\theta \qquad (\text{if } \Gamma \vdash e : \tau \text{ and } \Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta)$$

Calculating known representations and conventions, and the levity of a known convention:

$$\frac{\Gamma \vdash \tau : \mathsf{TYPE}\ \pi\ \nu \quad \pi \in PrimRep}{\Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi} \qquad \frac{\Gamma \vdash \tau : \mathsf{TYPE}\ \rho\ \eta \quad \eta \in KnownConv}{\Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta} \qquad \begin{aligned} lev(\mathtt{Eval}^\psi) &= \psi \\ lev(\mathtt{Call}[\overline{\pi}]) &= \mathsf{U} \end{aligned}$$

Fig. 8. Compiling $\mathcal{IL}$ to $\mathcal{ML}$

then we can evaluate the result of e directly by the main compilation translation, otherwise if it is $\mathtt{Call}[\overline{\pi}]$ then e must be called (not evaluated). To make sure that the definition and call sites of a primitive function match, we always fully η-expand these expressions when they are defined: either on the right-hand side of **let**s or in the body of `Closure`s. Because of η-expansion, this step of compilation is only defined when the calling convention is statically known (*e.g.,* it is not a variable n or partially-defined like $\mathtt{Call}[r_1, r_2, arity(n)]$). In any case, we next move to the main work-horse of compilation, $\mathcal{C}[\![e]\!]^\Gamma_\theta(\overline{a})$, that produces $\mathcal{ML}$ code to evaluate the result of e applied to the arguments $(\overline{a})$. Again, there are invariants to this translation that we will enumerate shortly.

For $\mathcal{C}[\![e]\!]^\Gamma_\theta(\overline{a})$, literal constants are just passed through, but compiling a call to `error` assumes precisely one argument. What if the user has written a partial application of `error`? Such partial applications are *always* η-expanded to be fully saturated, satisfying the requirement here. Compiling a variable x looks it up in the environment θ, but there is different $\mathcal{ML}$ code for *evaluating* x versus *calling* it, even when there are no arguments. In other words, a primitive function variable x with the empty calling convention `Call[]` compiles as $C[\![x]\!]^{\Gamma}_{[y_{\mathtt{PtrR}}/x]\theta}(\varepsilon) = y_{\mathtt{PtrR}}()$—a nullary function call—but a variable x with the convention $\mathtt{Eval}^{\gamma}$ compiles as to the pointer lookup $C[\![x]\!]^{\Gamma}_{[y_{\mathtt{PtrR}}/x]\theta}(\varepsilon) = y_{\mathtt{PtrR}}$.

Closure applications are compiled straightforwardly, and erasable arguments ϕ and binders $\Lambda\chi.e$ are simply dropped. $\mathcal{IL}$ answers can be compiled outright to a $\mathcal{ML}$ WHNF via $\mathcal{A}[\![A]\!]^{\Gamma}_{\theta}$. Compiling an application to an answer A depends on the nature of that answer:

- If $\mathcal{A}[\![A]\!]^{\Gamma}_{\theta}$ is a variable or constant (after type erasure), then it can be passed directly.
- Otherwise, name the argument with a **let** (respecting the A-normal form) and pass it by reference. In this case, the compiled argument will always have the form $\mathtt{I\#}(a)$ or $\mathtt{Clos}^{n}\, W$, meaning the **let**-binding will always be represented as a pointer into the heap.[15]

In the variable case, we do not need to track the levity or representation of the argument, because these decisions have already been made by the context, when the variable definition itself was compiled. Crucially, we did not have to look up any information in the typing environment to compile answer arguments; this is why no highlighted premises are needed in rule Fun-A-E.

In the case of an application $e\ e'$ to an arbitrary argument that needs to be computed, corresponding to Fun-E, we always generate a **let** similar to the second case for $e\ A$. However, for Fun-E, we need to determine the representation, convention, *and* levity of the binding, which could truly be anything. This corresponds to the highlighted side conditions in Fun-E.

6.3 Correctness of Compilation

Notice how the same polymorphism restrictions used in the typing rules also appear during compilation. Even though the defined compilation translation is partial (not every syntactically valid expression can be compiled), all well-typed $\mathcal{IL}$ expressions with a known convention have a defined compilation to $\mathcal{ML}$. In particular, $\mathcal{E}_{\eta}[\![e]\!]$ is well-defined for any closed expression $\vdash e : \tau : \mathsf{TYPE}\ \rho\ \eta$, where the syntax of known conventions η is given in Fig. 7.

In fact, we allow for a little more levity polymorphism during compilation: $\mathcal{E}_{\mathtt{Eval}^{g}}[\![e]\!]$, for a polymorphic levity g, is also allowed. That's because the generated code will be executed exactly when expression is evaluated: in other words, when a computation is forced, there is no difference between eager (U) or lazy (L). This added flexibility is essential for compiling levity polymorphic expressions appearing in strict contexts, such as in the discriminant of a **case** or first argument of App. Although implicit, the C translation assumes that evaluation of the compiled expression is being forced. In contrast, the $\mathcal{A}$ translation does not assume this, because it is used in contexts that do *not* force the expression. This small difference is how we are able to pass variables of *any* convention (eager, lazy, or primitive functions) without erroneously introducing extra strictness.

During compilation, we occasionally need to know representation (π) or convention (η) of a sub-expression. This appears in Fig. 8 as highlighted side conditions $\Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi$ and $\Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta$, respectively. In general, compilation could fail if the representation or convention in the kind of τ are partially unknown—that is, contains free variables. But any closed representation has the form π and any closed convention is equivalent to a η, as defined by the syntax of $\mathcal{ML}$ in Fig. 7. The places where this requirement appears corresponds exactly to the highlighted monomorphism restrictions in Fig. 3.

[15] A more feature-rich language may allow for representations other than just a single pointer, in this case. Even then, answers compile to values with syntactically manifest representations, so no additional typing information is needed here.

Theorem 3 (Closed Compilation). *If* $\vdash e : \tau$ *and* $\vdash \tau : \mathsf{TYPE}\,\rho\,\nu$ *then* $\mathcal{E}_\nu[\![e]\!]$ *is defined.*

This theorem just states when compilation succeeds in generating $\mathcal{ML}$ code from a closed $\mathcal{IL}$ expression. We should also expect that compilation preserves the behavior of that $\mathcal{IL}$ expression as well. In other words, if an expression is equal to some answer in $\mathcal{IL}$ (as per Fig. 4), then executing the compiled code should give the same answer in $\mathcal{ML}$. But we are not interested in evaluating primitive functions directly—they are called, not evaluated!—so answers will be of some `Eval`uatable types like (un)boxed integers, which are simple enough values to line up on the nose.

Theorem 4 (Soundness and Completeness).

(1) For any $\vdash e : \mathtt{Int\#}$, $\vdash e = i : \mathtt{Int\#}$ *if and only if* $\langle \mathcal{E}_{\mathtt{Eval}^{\mathsf{U}}}[\![e]\!] \mid \varepsilon \mid \varepsilon \rangle \mapsto^* \langle i \mid \varepsilon \mid H \rangle$.

(2) For any $\vdash e : \mathtt{Int}^{\gamma}$, $\vdash e = \mathtt{I\#}^{\gamma}\, i : \mathtt{Int}^{\gamma}$ *if and only if* $\langle \mathcal{E}_{\mathtt{Eval}^{\gamma}}[\![e]\!] \mid \varepsilon \mid \varepsilon \rangle \mapsto^* \langle \mathtt{I\#}(i) \mid \varepsilon \mid H \rangle$.

7 DYNAMIC ARITY

Consider the following program written in Haskell, where *exp* is some expensive function:[16]

$$\begin{array}{l} \mathbf{data}\,T = MkT\,(\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}) \\ t1 = MkT\;(\lambda x.\,\mathbf{let}\,z = exp\;x\,\mathbf{in}\,\lambda y.z + y) \\ t2 = MkT\;plus \end{array}
\qquad
\begin{array}{l} appT1 : T \to \mathtt{Int} \to \mathtt{Int} \\ appT1\,(MkT\,g) = g\,1 \\ appT2 : T \to \mathtt{Int} \\ appT2\,(MkT\,g) = g\,1\,2 \end{array}$$

In terms of the informal notion of arity in the source language (Section 2), we can say that $t1$ stores an arity 1 closure and $t2$ stores an arity 2 closure. Likewise, *appT1* performs an arity 1 application and *appT2* performs an arity 2 application. This can be compiled to $\mathcal{IL}$ (extended with data type declarations) similar to the translation in Fig. 5, which formalizes the arities like so:

$$\begin{array}{l} \mathbf{data}\,T = MkT\;{}^{\mathsf{L}}\{\mathtt{Int} \rightsquigarrow {}^{\mathsf{L}}\{\mathtt{Int} \rightsquigarrow \mathtt{Int}\}\} \\ t1 = MkT\;(\mathtt{Clos}^{\mathsf{L}}\,\lambda x.\,\mathbf{let}\,z = exp\;x\,\mathbf{in}\,\mathtt{Clos}^{\mathsf{L}}\,\lambda y.z + y) \\ t2 = MkT\;(\mathtt{Clos}^{\mathsf{L}}\,\lambda x.\mathtt{Clos}^{\mathsf{L}}\,\lambda y.plus\;x\;y) \end{array}
\qquad
\begin{array}{l} appT1 : T \rightsquigarrow {}^{\mathsf{L}}\{\mathtt{Int} \rightsquigarrow \mathtt{Int}\} \\ appT1\,(MkT\,g) = \mathsf{App}\,g\,1 \\ appT2 : T \rightsquigarrow \mathtt{Int} \\ appT2\,(MkT\,g) = \mathsf{App}\,(\mathsf{App}\,g\,1)\,2 \end{array}$$

Notice how *appT2* $(MkTg)$ does not apply g to both arguments at once; instead, it must evaluate g applied to 1 first, then apply the returned closure to 2 in a separate step. The two-step application process is *mandated* by the type of the *MkT* constructor, even though most of the time *MkT* will ultimately be used store a closure capable of accepting both arguments at once. This arity demotion can be seen in $t1$, where the binary *plus* function is wrapped up in a chain of two unary closures, as required by *MkT*. As a result, the call *appT1* $t2$ must pass 1 and 2 separately to *plus*, losing the opportunity for the faster binary calling convention that seemed possible in the source program.

We can attempt improve the performance of $t2$ and *appT2* with an alternate translation:

$$\begin{array}{l} \mathbf{data}\,T' = MkT'\;{}^{\mathsf{L}}\{\mathtt{Int} \rightsquigarrow \mathtt{Int} \rightsquigarrow \mathtt{Int}\} \\ t1' = MkT'\;(\mathtt{Clos}^{\mathsf{L}}\,\lambda x.\lambda y.\,\mathbf{let}\,z = exp\;x\,\mathbf{in}\,z + y) \\ t2' = MkT'\;(\mathtt{Clos}^{\mathsf{L}}\,plus) \end{array}
\qquad
\begin{array}{l} appT1' : T' \rightsquigarrow \mathtt{Int} \rightsquigarrow \mathtt{Int} \\ appT1'\;(MkT'\,g)\,x = \mathsf{App}\,g\,1\,x \\ appT2' : T' \rightsquigarrow \mathtt{Int} \\ appT2'\;(MkT'\,g) = \mathsf{App}\,g\,1\,2 \end{array}$$

Now, MkT' contains a closure of a single binary function. This way, $appT2'$ $t2'$ steps to the single binary application *plus* 1 2, passing both arguments at once to *plus*. But the type of MkT' makes it impossible to contain closures that memoize work when applied to only one argument—an unfortunate, unintended ramification of this "optimization." Unlike before, the closure inside $t1'$ recomputes *exp* x every time the second argument is given, instead of memoizing the result once the first argument is provided. Here, we make this fact syntactically explicit by η-expanding the

[16] The exact same example applies to OCaml as well by replacing L with U in translations into $\mathcal{IL}$ that follow.

definitions of $t1'$ and $appT1'$, but because $\rightsquigarrow$-types are fully extensional, the same recomputation will occur no matter what. For example, $map\ (appT1'\ t1')\ [1..1000]$ recomputes $exp\ 1$ for all 1000 elements of the list, whereas the previous $map\ (appT1\ t1)\ [1..1000]$ only computes $exp\ 1$ once.

The root of the problem is that the data type definition of T forces us to pick *one* type for the constructor MkT. Because $\mathcal{IL}$ statically links this type to an arity, we are thus forced to pick *one* arity for MkT closures that is used throughout the entire program. MkT may often be used to hold essentially binary function closures, but the computational effect of unary application of MkT closures can still be crucial in certain corners of a program—either for asymptotic complexity in Haskell or for side effects in OCaml.

Rather than enforcing *statically* that every call is exactly saturated, perhaps we could perform a *dynamic* check. Given a binary application, we could interrogate the function value to check its arity, and behave differently depending on whether that arity is 1 (an over-saturated call), 2 (exactly saturated, the fast case), or greater than 2 (unsaturated). This is, in fact, what GHC does today, and corresponds to the "unknown" function calls of Marlow and Peyton Jones [2004] which inspect the arities of closure values at runtime to choose the best calling convention. But how can $\mathcal{IL}$—with its statically-tracked notion of arity—accommodate dynamic arity checking?

The key is to allow for the primitive functions contained in closures to have a different arity than their call site, thus requiring a dynamic check on all Applications of closures. Applying too few arguments creates a partial application, and applying too many is broken down into several steps. More formally, the syntax of $\mathcal{ML}$ can be extended with partial applications $\mathtt{Clos}^n\ f(\overline{a})$, where f has been applied to arguments $\overline{a}$ so far and n is the number of remaining arguments expected before f can be called. Now, consider these extra rules for dynamic handling a runtime arity mismatch:

$$
\begin{array}{llr}
(Apply) & \langle \mathtt{Clos}^n\ W(\overline{a}) \mid \mathsf{App}(\overline{a}'); K \mid H\rangle \mapsto \langle W(\overline{a},\overline{a}') \mid K \mid H\rangle & (\text{if } |\overline{a}'| = n) \\
(PApp) & \langle \mathtt{Clos}^n\ W(\overline{a}) \mid \mathsf{App}(\overline{a}'); K \mid H\rangle \mapsto \langle \mathtt{Clos}^{n-|\overline{a}'|}\ W(\overline{a},\overline{a}') \mid K \mid H\rangle & (\text{if } |\overline{a}'| < n) \\
(OApp) & \langle \mathtt{Clos}^n\ W(\overline{a}) \mid \mathsf{App}(\overline{a}',\overline{a}''); K \mid H\rangle \mapsto \langle W(\overline{a},\overline{a}') \mid \mathsf{App}(\overline{a}''); K \mid H\rangle & (\text{if } |\overline{a}'| = n, |\overline{a}''| > 0)
\end{array}
$$

In practice, these extra rules for partial- and over-application lets us treat closure types like ${}^{\mathsf{L}}\{\mathtt{Int} \rightsquigarrow {}^{\mathsf{L}}\{\mathtt{Int} \rightsquigarrow \mathtt{Int}\}\}$ and ${}^{\mathsf{L}}\{\mathtt{Int} \rightsquigarrow \mathtt{Int} \rightsquigarrow \mathtt{Int}\}$ as the same, without endangering type safety due to arity mismatch. For example, this would eliminate the difference between the T and T' data types from before. With these dynamic arity checks, it is safe to call $appT1\ t2'$: this results in a partial application because the caller ($appT1$) only provides one argument to the binary closure ($t2'$). Likewise, it is safe to call $appT2'\ t1$: this results in an over application where the caller ($appT2'$) wants to pass two arguments to a unary closure ($t1$). Yet, in the cases where the optimal arities do match (like $appT1\ t1$ and $appT2'\ t2'$), the fastest calling convention is used at runtime. Therefore, the types T and T' can be used interchangeably, in some sense, as long as arities are checked at runtime. More generally, dynamic arity checks lets us safely equate these two closure types:

$$ {}^{\gamma}\{\overline{\tau} \rightsquigarrow {}^{\gamma}\{\sigma\}\} = {}^{\gamma}\{\overline{\tau} \rightsquigarrow \sigma\} $$

This could be formalized in $\mathcal{IL}$ as a type equality or a type-safe coercion [Breitner et al. 2016]: both types are represented identically at runtime as closure objects with some runtime arity count.

In the end, both known and unknown calls of Marlow and Peyton Jones [2004] can be captured in the intermediate language. The arity of a type $\tau \rightsquigarrow \sigma$ is known statically by its kind, and the program must provide the right number of arguments and binders. However, types like ${}^{\gamma}\{\tau \rightsquigarrow \sigma\}$ and ${}^{\gamma}\{\tau \rightsquigarrow {}^{\gamma}\{\sigma\}\}$, which both store additional arity information at runtime, can be freely interchanged at compile time, as long as the arities are checked at runtime. To be clear, this extension has a trade-off: the closures described here are subject to extra dynamic checks. It is possible that an implementation would want to have both statically checked closures and dynamically checked ones. We can accommodate both by simply having two different closure types (with their own

`Clos` and `App`). Then, an optimizing compiler, or an expert user, can select the one with the best performance for a particular part of a program.

8 RELATED WORK

The system presented in this paper is the culmination of several independent lines of work on expressing performance issues directly in an intermediate language. The underlying theme is to capture the low-level details of calling conventions as features of a higher-level functional language.

8.1 Representation and Levity in the Kinds

The idea of distinguishing (un)boxed and (un)lifted types goes back to Peyton Jones and Launchbury [1991]. That distinction has been static until recent work added levity polymorphism to the mix [Eisenberg and Peyton Jones 2017], and shown that its utility is greater than expected (see Section 7 of that work). However, Eisenberg and Peyton Jones [2017] conflates *levity polymorphism* and *representation polymorphism*. Our contribution separates the two completely, with applications that are polymorphic in one but not the other. One our main requirements is to generate only *one* piece of code for every polymorphic definition. Certain definitions that must be rejected, because compilation would depend on a choice made at runtime. An alternative approach by Dunfield [2015] accepts more uses of levity polymorphism at the cost of generating *different* code for each choice—an exponential blowup of code size in practice—which we avoid.

8.2 Optimizing Curried Functions

Previous work established methods for optimizing curried function calls dynamically at runtime, avoiding the overhead of naively calling $((f\ 1)\ 2)\ 3$ by passing one argument at a time. In practice, f will often expect all three arguments before doing any interesting work, so those calls should be fused when possible. Fusing can be done by pushing many arguments on the stack at once (the push/enter model) [Krivine 2007; Leroy 1990] or by evaluating the arity of closures (the eval/apply model) [Marlow and Peyton Jones 2004]. In this work, we capture this dynamic type of optimization within the syntax and types of programs, as described in Section 7.

8.3 Function Arity in Types

While there is performance to be gained by dynamically optimizing curried function calls at runtime, it is even better to optimize statically at compile time. Of course, this is easy to do when the compiler can find the definition of the called function [Marlow and Peyton Jones 2004]. This scheme is easily thwarted by higher-order functions, so a less syntactic approach—like one based on types—can be beneficial. Uncurrying—representing a function $a{\to}b{\to}c{\to}d$ as $(a,b,c){\to}d$—is an obvious place to start, and has been investigated before [Bolingbroke and Peyton Jones 2009; Dargaye and Leroy 2009; Hannan and Hicks 1998]. However, when polymorphism is brought into the picture, type quantification is irreparably fused with multi-arity functions; see [Downen et al. 2019, Section 8.1].

Following Downen et al. [2019], $\mathcal{IL}$ instead retains the curried form of function types. However, $\mathcal{IL}$ goes significantly beyond that work by supporting type polymorphism over arrow types (Section 4.3), and polymorphism over levities (Section 4.4) and conventions (Section 4.5). Another difference is that Downen et al. [2019] had two function arrows, $(\tau \xrightarrow{\gamma} \sigma)$ and $(\tau \rightsquigarrow \sigma)$, whereas $\mathcal{IL}$ has just one arrow $(\tau \rightsquigarrow \sigma)$, plus the closure type ${}^{\gamma}\{\tau\}$. The two are inter-convertible: we showed how to translate $\tau \xrightarrow{\gamma} \sigma$ to $\mathcal{IL}$ for either γ in Fig. 5, and in the other direction we have ${}^{\gamma}\{\tau\} = () \xrightarrow{\gamma} \tau$ with $(\mathtt{Clos}^{\gamma}\ e) = \lambda^{\gamma}().e$ and $(\mathtt{App}\ e) = e\ ()$. Note that, to make the analogy operationally exact, the unit type $()$ should be an unboxed, empty tuple (*i.e.,* represented as 0 arguments at runtime). The approach here has a greater economy of concepts, and a nice correspondence with `Int#` and $\mathtt{Int}^{\gamma}$. However, two function arrows might be better for a practical compiler.

8.4 The Glasgow Haskell Compiler

GHC already implements a rich kind system, including polymorphism over types, kinds, and representations. Indeed GHC goes further: instead of a stratified zoo of different things (types, kinds, representations, *etc.*) as in Fig. 1, they are all *types* [Weirich et al. 2013] kept separate by their *kinds*. This is a fantastic simplification, immediately allowing polymorphism over all these conceptually-different things. This does, however, make it hard *not* to have polymorphism! Returning to Section 4.3, it would be hard to prevent instantiation of a forall-quantified type variable with an arrow type, requiring a restriction like "this quantified variable can have any kind *other than* `Call`." So GHC's infrastructure strongly encourages fully-fledged polymorphism.

8.5 Logical Foundations

The $\mathcal{IL}$ language is not an ad-hoc collection of design compromises driven by only performance considerations. Rather, it grows directly from principled foundations in logic.

Previous work on unboxed types and extensional functions shares the observation that *lifting*—in the sense of denotational semantics—corresponds to a *mismatch* between machine primitives and the semantics of a programming language. Unlifted types can be implemented more directly—and therefore more efficiently—in a machine. But the cause of lifting depends on the type: unlifted integers need to be call-by-value whereas unlifted curried functions need to be call-by-name. The first reconciliation was achieved in call-by-push-value [Levy 2001], which avoids all lifting unless explicitly requested. As such, this paper can be seen as a practical extension of this foundation.

The same connection between types and evaluation is also tied to *focusing and polarity* [Andreoli 1992; Laurent 2002] in proof search, which corresponds to pattern-matching in functional programming [Zeilberger 2008, 2009] and semantics and computation [Munch-Maccagnoni 2009, 2013]. Recently, these mixed evaluation strategy languages have been extended with practical features like call-by-need evaluation [Downen and Ariola 2018; McDermott and Mycroft 2019] to model shared computation. Of note, the types in $\mathcal{IL}$ used for boxing and indirection correspond exactly to the "polarity shifts" of Downen and Ariola [2018] to and from call-by-need. In particular, the boxed integer type corresponds to an "up shift" ($\mathtt{Int} = \Uparrow\mathtt{Int\#}$) and the function closures to a "down shift" ($\{\tau \rightsquigarrow \sigma\} = \Downarrow(\tau \rightsquigarrow \sigma)$). For the sake of usability, $\mathcal{IL}$ performs other implicit polarity conversions of types based on their context. For example, closing over a non-function type like `Int#` implicitly shifts it to a "nullary function" (there written $\uparrow \mathtt{Int}$), expressed by the encoding $\{\mathtt{Int\#}\} = \Downarrow\uparrow\mathtt{Int\#}$.

9 CONCLUSION

This paper illustrates a cohesive system for including low-level details—specifically *representation*, *levity*, and *arity*—inside a higher-level intermediate language. Not only does this let the language express intensional properties of programs, it also lets programs *abstract* over these details when they do not impact compilation. Parts of this work have been implemented already in the Glasgow Haskell Compiler, and we intend to further implement the entirety of kinds as calling conventions. The story presented here takes an explicitly typed intermediate language and—through type-driven elaboration—compiles it to an untyped target language. As future work, it could be enlightening to consider how types might be preserved by compilation by giving a sufficiently expressive type system for the lower-level language. Since the main objective is to capture performance in the intermediate language, we would also like to characterize the cost of computation in its semantics.

ACKNOWLEDGMENTS

The authors would like to thank Norman Ramsey and the anonymous reviewers, whose feedback was invaluable in improving the presentation of this paper. The material is based upon work supported by the National Science Foundation under Grant No. 1719158.

REFERENCES

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

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

Maximilian C. Bolingbroke and Simon L. Peyton Jones. 2009. Types Are Calling Conventions. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, 1–12.

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.

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016), e15. https://doi.org/10.1017/S0956796816000150

Zaynah Dargaye and Xavier Leroy. 2009. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation* 22, 3 (2009), 199–231.

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.

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 (Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 58–70. https://doi.org/10.1145/3331545.3342594

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.

Richard Eisenberg. 2019. GHC Proposal 29: revised levity polymorphism. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0029-levity-polymorphism.rst

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*. 525–539.

Sebastian Graf. 2020. GHC Proposal 265: unlifted data types. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0265-unlifted-datatypes.rst

John Hannan and Patrick Hicks. 1998. Higher-Order unCurrying. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 1–11.

Jean-Louis Krivine. 2007. A Call-By-Name Lambda-Calculus Machine. *Higher-Order and Symbolic Computation* 20, 3 (2007), 199–207.

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

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

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

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.

Andrew Martin. 2019a. GHC Proposal 112: unlifted arrays. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0112-unlifted-array.rst

Andrew Martin. 2019b. GHC Proposal 203: pointer rep. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0203-pointer-rep.rst

Andrew Martin. 2019c. GHC Proposal 98: unlifted newtypes. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0098-unlifted-newtypes.rst

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.

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

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

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.

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.

Amr Sabry and Matthias Felleisen. 1993. Reasoning About Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3-4 (Nov. 1993), 289–360.

Amr Sabry and Philip Wadler. 1997. A Reflection on Call-by-Value. *ACM Transactions on Programming Languages and Systems* 19, 6 (1997), 916–941.

Alex Theriault. 2019. GHC Proposal 209: levity-polymorphic Lift. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0209-levity-polymorphic-lift.rst

Philip Wadler, Walid Taha, and David Macqueen. 1998. How to add laziness to a strict language without even being odd. In *Proceedings of the Standard ML Workshop*.

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *International Conference on Functional Programming (ICFP '13)*. ACM.

Noam Zeilberger. 2008. On the Unity of Duality. *Annals of Pure and Applied Logic* 153, 1 (2008), 660–96.

Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching*. Ph.D. Dissertation. Carnegie Mellon University.

A OPERATIONAL SEMANTICS FOR $\mathcal{IL}$

A.1 Notation

We use $\mapsto$ to denote the single step relation of an operational semantics, $\mapsto^{?}$ to denote its reflexive closure, $\mapsto^{+}$ to denote its transitive closure, and $\mapsto^{*}$ to denote its reflexive-transitive closure. This same notation is used for other reduction arrows.

A.2 Substitution-based (call-by-name) operational semantics

$$
\begin{array}{rl}
\mathit{Final} \ni \mathit{Fin} ::= & V \mid E[\mathtt{error}\ \rho\ \gamma\ \tau\ V] \\
\mathit{Value} \ni \quad V ::= & x\ \overline{\phi} \mid c\ \overline{\phi} \mid \mathtt{I\#}^{\gamma}\ V \mid \mathtt{Clos}^{\gamma}\ e \mid \lambda x{:}\tau.e \mid \Lambda\chi.V \\
\mathit{EvalCxt} \ni \quad E ::= & \square \mid e\ E \mid E\ e \mid \mathsf{App}\ E \mid \mathtt{I\#}^{\gamma}\ E \mid \mathbf{case}\ E\ \mathbf{of}\ \mathtt{I\#}\ x \to e \mid E\ \phi \mid \Lambda\chi.E
\end{array}
$$

Note that the definition of *value* above only differs from *answer* (in Fig. 1) in that we don't allow for $\beta_\forall$ redexes. In other words, every answer reduces (by the following operational semantics) to some value: for all answers A there is a value V such that $A \mapsto_{\beta_\forall}^{*} V$. A the correspondingly generalized notion of substitutable value can be bound to a variable is determined in part by the type of the expression as given by the following rules:

$$
\frac{}{\Gamma \vdash V\ \mathbf{val}}
\qquad
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \mathsf{TYPE}\ \rho\ \mathtt{Call}[\alpha]}{\Gamma \vdash e\ \mathbf{val}}
\qquad
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \mathsf{TYPE}\ \rho\ \mathtt{Eval}^{\mathsf{L}}}{\Gamma \vdash e\ \mathbf{val}}
$$

The decomposition of an expression into an evaluation context surrounding an expression is also defined in part by typing information, written with the judgement $\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$, by the following inference rules:

$$
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \square \mathbin{@} e : \tau \Longrightarrow \tau}
\qquad
\frac{\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \mathtt{Int\#}}{\Gamma \vdash \mathtt{I\#}^{\gamma}\ E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \mathtt{Int}^{\gamma}}
\qquad
\frac{\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \mathtt{Int}^{\gamma} \quad \Gamma, x : \mathtt{Int\#} \vdash e' : \sigma}{\Gamma \vdash \mathbf{case}\ E\ \mathbf{of}\ \mathtt{I\#}\ x \to e' \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}
$$

$$
\frac{\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} {}^{\gamma}\{\sigma\}}{\Gamma \vdash \mathsf{App}\ E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}
\qquad
\frac{\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma}{\Longrightarrow} \forall\chi.\sigma}{\Gamma \vdash E\ \phi \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma[\phi/\chi]}
\qquad
\frac{\Gamma, \chi \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}{\Gamma \vdash \Lambda\chi.E \mathbin{@} e : \tau \overset{\Gamma',\chi}{\Longrightarrow} \forall\chi.\sigma}
$$

$$
\frac{\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \tau' \quad \Gamma \vdash \tau' : \mathsf{TYPE}\ \rho\ \mathtt{Eval}^{\mathsf{U}} \quad \Gamma \vdash e : \tau' \rightsquigarrow \sigma \quad \vdash \rho\ \mathbf{rep}}{\Gamma \vdash e'\ E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}
$$

$$
\frac{\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \tau' \rightsquigarrow \sigma \quad \Gamma \vdash e' : \tau' \quad \Gamma \vdash e'\ \mathbf{val} \quad \Gamma \vdash \tau'\ \mathbf{mono\text{-}rep} \quad \Gamma \vdash \tau'\ \mathbf{mono\text{-}conv}}{\Gamma \vdash E\ e' \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}
$$

The (typed) operational stepping relation is written $\Gamma \vdash e \mapsto e' : \sigma$, and always assumes as a precondition that $\Gamma \vdash e : \sigma$ holds. The primary reduction steps are by the following rules (where we omit the typing precondition on the left-hand side):

$$
\begin{array}{rcl}
(\beta_{\rightsquigarrow}) & \Gamma \vdash (\lambda x{:}\tau.e)\ e' \mapsto e[e'/x] : \sigma & (\text{if } \Gamma \vdash e'\ \mathbf{val}) \\
(\beta_{\forall}) & \Gamma \vdash (\Lambda\chi.V)\ \phi \mapsto V[\phi/\chi] : \sigma[\phi/\chi] & \\
(\beta_{\{\}}) & \Gamma \vdash \mathsf{App}\ (\mathtt{Clos}^{\gamma}\ e) \mapsto e : \sigma & \\
(\beta_{\mathtt{Int}}) & \Gamma \vdash \mathbf{case}\ \mathtt{I\#}^{\gamma}\ V\ \mathbf{of}\ \mathtt{I\#}\ x \to e \mapsto e[V/x] : \sigma &
\end{array}
$$

Additionally, the operational steps are closed under evaluation contexts:

$$
\frac{\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma \quad \Gamma, \Gamma' \vdash e \mapsto e' : \tau}{\Gamma \vdash E[e] \mapsto E[e'] : \sigma}\ \mathit{compat}
$$

Lemma 1 (Typed Decomposition). *If* $\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ *then* $\Gamma, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash E[e] : \sigma$.

Proof. By induction on the derivation of the decomposition $\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$. □

Lemma 2 (Unique Decomposition). *For every* $\mathcal{IL}$ *expression* $\Gamma \vdash e : \sigma$, *either:*

(1) $e \in$ *Value, or*

(2) there is a unique $\Gamma \vdash E \mathbin{@} e' : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ *such that either*

(a) e' *is a variable or constant, or*

(b) $\Gamma \vdash e' \mapsto e'' : \tau$ *directly (i.e., not by compat).*

Proof. By induction on the typing derivation of $\Gamma \vdash e : \sigma$. □

Corollary 1 (Determinism). *If* $\Gamma \vdash e \mapsto e_1 : \sigma$ *and* $\Gamma \vdash e \mapsto e_2 : \sigma$ *then* $e_1 =_\alpha e_2$.

Proof. Follows directly from Lemma 2 and the fact that the operational rules don't overlap. □

Lemma 3 (Stability under Substitution).

(1) If $\Gamma \vdash e'' : \sigma$ *and* $\Gamma \vdash e''$ **val** *then* $\Gamma, x : \sigma \vdash e \mapsto e' : \tau$ *implies* $\Gamma \vdash e[e''/x] \mapsto e'[e''/x] : \sigma$.

(2) If $\Gamma \vdash [\phi/\chi]$ **poly** *then* $\Gamma, \chi \vdash e \mapsto e' : \tau$ *if and only if* $\Gamma[\phi/\chi] \vdash e[\phi/\chi] \mapsto e'[\phi/\chi] : \tau[\phi/\chi]$.

Similarly, the decomposition of evaluation contexts E *is stable under substitution.*

Proof. By induction on the derivation of the reduction, or the syntax of the context. □

Lemma 4 (Typed Substitution). *(1) If* $\Gamma, x : \sigma \vdash e : \tau$ *and* $\Gamma \vdash e' : \sigma$ *then* $\Gamma \vdash e[e'/x] : \tau$.

(2) If $\Gamma, t : \kappa \vdash e : \tau$ *and* $\Gamma \vdash \sigma : \kappa$ *then* $\Gamma \vdash e[\sigma/t] : \tau[\sigma/t]$.

(3) If $\Gamma, g \vdash e : \tau$ *and* $\Gamma \vdash \gamma$ **lev** *then* $\Gamma[\gamma/g] \vdash e[\gamma/g] : \tau[\gamma/g]$.

(4) If $\Gamma, r \vdash e : \tau$ *and* $\Gamma \vdash \rho$ **rep** *then* $\Gamma[\rho/r] \vdash e[\rho/r] : \tau[\rho/r]$.

(5) If $\Gamma, n \vdash e : \tau$ *and* $\Gamma \vdash \nu$ **conv** *then* $\Gamma[\nu/n] \vdash e[\nu/n] : \tau[\nu/n]$.

Proof. By induction on the given typing derivation of e. □

Lemma 5 (Progress). *If* $\vdash e : \tau$ *then either:*

(1) $e \in$ *Final, or*

(2) $\vdash e \mapsto e' : \tau$.

Proof. Follows directly from Lemma 2 and compatibility. □

Lemma 6 (Preservation). *If* $\Gamma \vdash e \mapsto e' : \tau$ *then* $\Gamma \vdash e' : \tau$.

Proof. Follows by induction on the decomposition of expressions into evaluation contexts and redexes and cases on the operational rules, using the fact that well-typed substitution preserves typing (Lemma 4). □

A.3 Environment-based (call-by-need) operational semantics

The primary difference between the call-by-need operational semantics—using environments to handle variables—versus the call-by-name operational semantics—using substitution instead—is *sharing*: when a lazy expression is bound in call-by-need, it is computed *at most once*. In order to make this distinction more syntactically apparent, here we officially at **let**-expressions to the language, with the typing rules given by the encoding

$$\textbf{let}\, x{:}\tau = e \,\textbf{in}\, e' \triangleq (\lambda x{:}\tau.e')\ e$$

The refined definition of final expressions, evaluation contexts, *etc.*is given as follows:

$$
\begin{array}{rrl}
\mathit{Value} \ni & V & ::= c\ \overline{\phi} \mid R \mid \Lambda\chi.V \\
\mathit{Reference} \ni & R & ::= \mathtt{I\#}^{\gamma}\ a \mid \mathtt{Clos}^{\gamma}\ e \mid \lambda x{:}\tau.e \mid \Lambda\chi.R \\
\mathit{Arg} \ni & a & ::= c\ \overline{\phi} \mid x\ \overline{\phi} \mid \Lambda\chi.a \\
\mathit{Final} \ni & \mathit{Fin} & ::= H[V] \mid E[\mathtt{error}\ \rho\ \gamma\ \tau\ a] \\
\mathit{EvalCxt} \ni & E & ::= \square \mid F[E] \mid B[E] \\
\mathit{ClosCxt} \ni & H & ::= \square \mid B[H] \\
\mathit{FrameCxt} \ni & F & ::= \square\ a \mid \mathsf{App}\ \square \mid \mathbf{case}\ \square\ \mathbf{of}\ \mathtt{I\#}\ x \to e \mid \mathbf{let}\ x{:}\tau = \square\ \mathbf{in}\ e \mid \square\ \phi \\
\mathit{BindCxt} \ni & B & ::= \mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ \square \mid \Lambda\chi.\square
\end{array}
$$

Notice that to support call-by-need evaluation, we further distinguish *references* to values, which may be copied at will, versus *bindable* expressions, which are merely allocated. Intuitively, referenced values may be copied, whereas bound expressions must be fully evaluated first before they can be copied.

$$
\frac{}{\Gamma \vdash R\ \mathbf{ref}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathtt{Call}[\alpha]}{\Gamma \vdash e\ \mathbf{ref}}
$$

$$
\frac{\Gamma \vdash e\ \mathbf{ref}}{\Gamma \vdash e\ \mathbf{bind}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Eval}^{\mathsf{L}}}{\Gamma \vdash e\ \mathbf{bind}}
$$

The type-driven decomposition of expressions into evaluation contexts and their sub-expression is the same for all syntactic forms except for applications and **let**-expressions (which are new for call-by-need). Decomposition for these cases are defined as:

$$
\frac{\Gamma \vdash E @ e : \tau \overset{\Gamma'}{\Longrightarrow} \tau' \rightsquigarrow \sigma \quad \Gamma \vdash a : \tau'}{\Gamma \vdash E\ a @ e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}
$$

$$
\frac{\Gamma, x : \tau' \vdash E @ e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma \quad \Gamma \vdash e' : \tau' \quad \Gamma \vdash e'\ \mathbf{bind} \quad \boxed{\Gamma \vdash \tau'\ \mathbf{mono\text{-}rep}}}{\Gamma \vdash \mathbf{let}\ x{:}\tau' = e'\ \mathbf{in}\ E @ e : \tau \overset{\Gamma', x:\tau'}{\Longrightarrow} \sigma}
$$

$$
\frac{\Gamma \vdash E @ e : \tau \overset{\Gamma'}{\Longrightarrow} \tau' \quad \Gamma, x : \tau' \vdash e' : \sigma \quad \Gamma \vdash \tau' : \mathsf{TYPE}\ \rho\ \mathsf{Eval}^{\mathsf{U}} \quad \boxed{\Gamma \vdash \tau'\ \mathbf{mono\text{-}rep}}}{\Gamma \vdash \mathbf{let}\ x{:}\tau' = E\ \mathbf{in}\ e' @ e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}
$$

$$
\frac{\Gamma, x : \tau' \vdash E_1 @ x : \tau' \overset{\Gamma_1}{\Longrightarrow} \sigma \quad \Gamma \vdash E_2 @ e : \tau \overset{\Gamma_2}{\Longrightarrow} \tau' \quad \Gamma \vdash \tau' : \mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Eval}^{\mathsf{L}}}{\Gamma \vdash \mathbf{let}\ x{:}\tau' = E_2\ \mathbf{in}\ E_1[x] @ e : \tau \overset{\Gamma_2}{\Longrightarrow} \sigma}
$$

As with decomposing an evaluation context, we also need to confirm that final results are properly decomposed into the appropriate context surrounding either a value or an error:

$$
\frac{\Gamma \vdash H @ V : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}{\Gamma \vdash H[V]\ \mathbf{fin}} \qquad \frac{\Gamma \vdash E @ \mathtt{error}\ \rho\ \gamma\ \tau\ a : \tau \overset{\Gamma'}{\Longrightarrow} \sigma}{\Gamma \vdash E[\mathtt{error}\ \rho\ \gamma\ \tau\ a]\ \mathbf{fin}}
$$

The reduction rules for call-by-need are also updated from the call-by-name semantics. Of note, $\beta_{\rightsquigarrow}$ is much more restricted to only substituting arguments which (after type erasure) are variables or constants. The analogous substitution is repeated for renaming in a **let**-expression.

$$
\begin{array}{rll}
(\beta_{\leadsto}) & \Gamma \vdash (\lambda x{:}\tau.e)\ a \mapsto e[a/x] : \sigma & \\
(\beta_{\forall}) & \Gamma \vdash (\Lambda \chi.\mathit{Fin})\ \phi \mapsto \mathit{Fin}[\phi/\chi] : \sigma[\phi/\chi] & \\
(\beta_{\{\}}) & \Gamma \vdash \mathtt{App}\ (\mathtt{Clos}^{\gamma}\ e) \mapsto e : \sigma & \\
(\beta_{\mathtt{Int}}) & \Gamma \vdash \mathbf{case}\ \mathtt{I\#}^{\gamma}\ a\ \mathbf{of}\ \mathtt{I\#}\ x \to e \mapsto e[a/x] : \sigma & \\
(\mathit{rename}) & \Gamma \vdash \mathbf{let}\, x{:}\tau = a\ \mathbf{in}\, e \mapsto e[a/x] : \sigma & \\
(\mathit{name}_{\mathtt{I\#}}) & \Gamma \vdash \mathtt{I\#}^{\gamma}\ e \mapsto \mathbf{let}\, x{:}\mathtt{Int\#} = e\ \mathbf{in}\ \mathtt{I\#}^{\gamma}\ x : \mathtt{Int}^{\gamma} & (\text{if } e \notin \mathit{Arg}) \\
(\mathit{name}) & \Gamma \vdash e'\ e \mapsto \mathbf{let}\, x{:}\tau = e\ \mathbf{in}\, e'\ x : \sigma & (\text{if } \Gamma \vdash e : \tau \text{ and } e \notin \mathit{Arg})
\end{array}
$$

Because the grammar of evaluation context is simplified to be primarily based on **let**s—rather than chains of curried function application—we convert more complex applications to an alternative **let** form, as done by the *name* and $\mathit{name}_{\mathtt{I\#}}$ rules. These **let**-expressions are interpreted by variable *look*up (which inlines the definition of a **let** only when its needed in an evaluation context, and only when the definition is a copyable reference). The presence of delayed **let** bindings also necessitates administrative *comm*uting conversions, which push bindings out of the way to bring frames of an evaluation context closer to the root of a final expression.

$$
\frac{\Gamma \vdash E \mathbin{@} x : \tau \overset{\Gamma'}{\Longrightarrow} \sigma \quad x \notin \mathrm{dom}(\Gamma) \quad \Gamma \vdash e\ \mathbf{ref}}{\Gamma \vdash \mathbf{let}\, x{:}\tau = e\ \mathbf{in}\, E[x] \mapsto \mathbf{let}\, x{:}\tau = e\ \mathbf{in}\, E[e] : \sigma}\ \mathit{look}
$$

$$
\frac{\Gamma \vdash F \mathbin{@} \mathbf{let}\, x{:}\tau' = e\ \mathbf{in}\, \mathit{Fin} : \tau \Longrightarrow \sigma \quad \Gamma \vdash \mathbf{let}\, x{:}\tau' = e\ \mathbf{in}\, \mathit{Fin}\ \mathbf{fin}}{\Gamma \vdash F[\mathbf{let}\, x{:}\tau' = e\ \mathbf{in}\, \mathit{Fin}] \mapsto \mathbf{let}\, x{:}\tau' = e\ \mathbf{in}\, F[\mathit{Fin}] : \sigma}\ \mathit{comm}
$$

$$
\frac{\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma \quad \Gamma, \Gamma' \vdash e \mapsto e' : \tau}{\Gamma \vdash E[e] \mapsto E[e'] : \sigma}\ \mathit{compat}
$$

Lemma 7 (Typed Decomposition). *If* $\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ *then* $\Gamma, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash E[e] : \sigma$.

Proof. By induction on the derivation of the decomposition $\Gamma \vdash E \mathbin{@} e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$. □

Lemma 8 (Unique Decomposition). *For every* $\mathcal{IL}$ *expression* $\Gamma \vdash e : \sigma$*, either:*

(1) $e \in \mathit{Final}$*, or*

(2) there is a unique $\Gamma \vdash E \mathbin{@} e' : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ *such that either*

   (a) e' *is a variable or constant, or*

   (b) $\Gamma \vdash e' \mapsto e'' : \tau$ *directly (i.e., not by compat).*

Proof. By induction on the typing derivation of $\Gamma \vdash e : \sigma$. □

Corollary 2 (Determinism). *If* $\Gamma \vdash e \mapsto e_1 : \sigma$ *and* $\Gamma \vdash e \mapsto e_2 : \sigma$ *then* $e_1 =_\alpha e_2$.

Proof. Follows directly from Lemma 8 and the fact that the operational rules don't overlap. □

Lemma 9 (Progress). *If* $\vdash e : \tau$ *then either:*

(1) $e \in \mathit{Final}$*, or*

(2) $\vdash e \mapsto e' : \tau$.

Proof. Follows directly from Lemma 8 and compatibility. □

Lemma 10 (Preservation). *If* $\Gamma \vdash e \mapsto e' : \tau$ *then* $\Gamma \vdash e' : \tau$.

Proof. Follows by induction on the decomposition of expressions into evaluation contexts and redexes and cases on the operational rules. □

A.4 Bisimulation between call-by-name and call-by-need

Definition 1 (Unwinding Simulation). The base simulation relation between the call-by-need and call-by-name operational semantics of $\mathcal{IL}$, written $\Gamma \vdash_{\mathcal{U}} e \sim e' : \tau$ where e may contain **let**-expressions but e' cannot, is defined inductively by the following rules for unwinding **let**s:

$$\frac{\Gamma \vdash_{\mathcal{U}} e_e \sim e_s : \tau \quad \Gamma, \overline{y_i : \tau} \vdash_{\mathcal{U}} e'_e \sim e'_s : \sigma \quad \Gamma \vdash e_s \ \mathbf{val} \quad \Gamma \vdash e_{si} \mapsto^* e_s : \tau}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau = e_e \,\mathbf{in}\, e'_e[\overline{x/y_i}] \sim e'_s[\overline{e_{si}/y_i}] : \sigma} \ \mathit{copy}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} e_e \sim e_s : \tau \quad \Gamma, x : \tau \vdash_{\mathcal{U}} e'_e \sim e'_s : \sigma}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau = e_e \,\mathbf{in}\, e'_e \sim (\lambda x{:}\tau.e'_e)\ e_s : \sigma} \ \mathit{share}$$

plus rules for compatibility with all other syntactic forms. The full simulation is then defined by inlining some unneeded names in the call-by-name semantics:

$$\frac{\Gamma \vdash_{\mathcal{U}} e \sim e'' : \tau \quad \Gamma \vdash e'' \mapsto^*_{\mathit{unname},\mathit{unname}_{\mathtt{I\#}}} e' : \tau}{\Gamma \vdash_{\mathcal{N}} e \sim e' : \tau}$$

$$\begin{array}{lll} (\mathit{unname}) & \Gamma \vdash (\lambda x{:}\tau.e'\ x)\ e \mapsto e'\ e : \sigma & (\text{if } x \notin \mathrm{FV}(e')) \\ (\mathit{unname}_{\mathtt{I\#}}) & \Gamma \vdash (\lambda x{:}\mathtt{Int\#}.\mathtt{I\#}^{\gamma}\ x)\ e \mapsto \mathtt{I\#}^{\gamma}\ e : \mathtt{Int}^{\gamma} & \end{array}$$

Lemma 11 (Context Unwinding). *(1) If $\Gamma \vdash E_e \ @\ e_e : \tau \overset{\Gamma',\Gamma''}{\Longrightarrow} \sigma$ in the call-by-need $\mathcal{IL}$, $\Gamma \vdash_{\mathcal{U}} E_e[e_e] \sim e''_s : \sigma$, and $\Gamma \nvdash e_e$ **val**, then $\Gamma \vdash e''_s \mapsto^* E_s[e_s[\theta]] : \tau$ for some evaluation context E_s and substitution $\theta = [\overline{e/x}]$ in the call-by-name $\mathcal{IL}$ such that $\Gamma \vdash E_s \ @\ e_s[\theta] : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ and $\Gamma, \Gamma', \Gamma'' \vdash_{\mathcal{U}} e_e \sim e_s : \tau$. Furthermore, if $\Gamma, \Gamma', \Gamma'' \vdash e_e \mapsto^* e'_e : \tau$ and $\Gamma, \Gamma', \Gamma'' \vdash e_s \mapsto^* e'_s : \tau$ such that $\Gamma, \Gamma', \Gamma'' \vdash e'_e \sim e'_s : \tau$, then $\Gamma E_e[e'_e] \sim E_s[e'_s[\theta]] : \tau$.*

*(2) If $\Gamma \vdash H_e \ @\ e_e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ and $\Gamma \vdash e_e$ **val** in the call-by-need $\mathcal{IL}$ and $\Gamma \vdash_{\mathcal{U}} H_e[e_e] \sim e'_s : \sigma$, then $\Gamma \vdash e'_s \mapsto^* e_s[\theta] : \tau$ for some substitution $\theta = [\overline{e/x}]$ in the call-by-name $\mathcal{IL}$ such that $\Gamma, \Gamma' \vdash_{\mathcal{U}} e_e \sim e_s : \tau$.*

Proof. By simultaneous induction on the evaluation context decomposition and the unwinding simulation (note that the part (2) follows analogously to part (1) for bound **let**s):

- The cases for all non-**let** evaluation contexts follow from the inductive hypothesis.
- $\Gamma \vdash \mathbf{let}\, x{:}\tau' = e'_e \,\mathbf{in}\, E_e \ @\ e_e : \tau \overset{\Gamma',\Gamma'',x:\tau'}{\Longrightarrow} \sigma$ because $\Gamma, x : \tau' \vdash E_e \ @\ e_e : \tau \overset{\Gamma',\Gamma''}{\Longrightarrow} \sigma$ and $\Gamma \vdash e'_e$ **bind**. This case depends on the rule of the simulation:
  - *share*: We have

$$\frac{\Gamma, x : \tau' \vdash_{\mathcal{U}} E_e[e_e] \sim e_s : \sigma \quad \Gamma \vdash_{\mathcal{U}} e'_e \sim e'_s : \tau'}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = e'_e \,\mathbf{in}\, E_e[e_e] \sim (\lambda x{:}\tau'.e_s)\ e'_s : \sigma}$$

By the inductive hypothesis, we get that

$$\Gamma \vdash e_s \mapsto^* E_s[e_{s1}[\theta]] : \sigma \quad \Gamma, x : \tau' \vdash E_s \ @\ e_{s1}[\theta] : \tau \overset{\Gamma'}{\Longrightarrow} \sigma \quad \Gamma, x : \tau', \Gamma', \Gamma'' \vdash_{\mathcal{U}} e_e \sim e_{s1} : \tau$$

In other words, we reduce to

$$(\lambda x{:}\tau'.e_s)\ e'_s \mapsto_{\beta_{\leadsto}} e_s[e'_s/x] \mapsto^* E_s[e'_s/x][e_{s1}[\theta, e'_s/x]]$$

which gives our new evaluation context and substitution, since evaluation contexts are stable under substitution (Lemma 3). Any further reduction of e_e and e_{s1} preserve the relation by the *copy* rule.

– *copy*: We have $E_{e0}[e_{e0}][\overline{x/y_i}] = E_e[e_e]$ such that

$$\frac{\Gamma, \overline{y_i : \tau'} \vdash_{\mathcal{U}} E_{e0}[e_{e0}] \sim e_s : \sigma \quad \Gamma \vdash_{\mathcal{U}} e'_e \sim e'_s : \tau' \quad \Gamma \vdash e'_{si} \mapsto e'_s : \tau'}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = e'_e\, \mathbf{in}\, E_{e0}[e_{e0}][\overline{x/y_i}] \sim e_s[\overline{e'_{si}/y_i}] : \sigma}$$

By the inductive hypothesis, we get that

$$\Gamma, \overline{y_i : \tau'} \vdash e_s \mapsto^* E_{s0}[e_{s0}[\theta]] : \sigma \quad \Gamma, \overline{y_i : \tau'} \vdash E_{s0} \mathbin{@} e_{s0}[\theta] : \tau \overset{\Gamma'}{\Longrightarrow} \sigma \quad \Gamma, \overline{y_i : \tau'} \vdash e_{e0} \sim e_{s0} : \sigma$$

In other words, by stability of call-by-name reduction under substitution (Lemma 3), we reduce to $e_s[\overline{e'_{si}/y_i}] \mapsto^* E_{s0}[e_{s0}[\theta, \overline{e'_{si}/y_i}]]$. Any further reduction of e_e and e_{s0} preserve the relation by the *copy* rule.

- $\Gamma \vdash \mathbf{let}\, x{:}\tau' = E_e\, \mathbf{in}\, e'_e \mathbin{@} e_e : \tau \overset{\Gamma', \Gamma''}{\Longrightarrow} \sigma$ because $\Gamma \vdash E_e \mathbin{@} e_e : \tau \overset{\Gamma', \Gamma''}{\Longrightarrow} \tau'$ and $\Gamma \vdash \tau' : \mathtt{TYPE}\ \rho\ \mathtt{Eval}^{\mathtt{U}}$. Only the *share* rule applies, so we have that

$$\frac{\Gamma, x : \tau' \vdash_{\mathcal{U}} e'_e \sim e'_s : \sigma \quad \Gamma \vdash_{\mathcal{U}} E_e[e_e] \sim e_s : \tau'}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = E_e[e_e]\, \mathbf{in}\, e'_e \sim (\lambda x{:}\tau'.e'_s)\ e_s : \sigma}$$

By the inductive hypothesis, we get that

$$\Gamma \vdash e_s \mapsto^* E_{s0}[e_{s0}[\theta]] : \tau' \qquad \Gamma \vdash E_{s0} \mathbin{@} e_{s0}[\theta] : \tau \overset{\Gamma'}{\Longrightarrow} \sigma \qquad \Gamma, \Gamma', \Gamma'' \vdash_{\mathcal{U}} e_e \sim e_{s0} : \tau$$

In other words, we reduce to $(\lambda x{:}\tau'.e'_s)\ e_s \mapsto^* (\lambda x{:}\tau'.e'_s)\ E_{s0}[e_{s0}[\theta]]$. Any further reduction of e_e and e_{s0} preserve the relation by the *share* rule.

- $\Gamma \vdash \mathbf{let}\, x{:}\tau' = E_{e2}\, \mathbf{in}\, E_{e1}[x] \mathbin{@} e_e : \tau \overset{\Gamma_2, \Gamma''}{\Longrightarrow} \sigma$ because $\Gamma \vdash E_{2e} \mathbin{@} e_e : \tau \overset{\Gamma_2, \Gamma''}{\Longrightarrow} \tau'$ and $\Gamma \vdash E_{1e} \mathbin{@} x : \tau' \overset{\Gamma_2, \Gamma'', x:\tau'}{\Longrightarrow} \sigma$ and $\Gamma \vdash \tau' : \mathtt{TYPE}\ \rho\ \mathtt{Eval[L]}$. This case depends on the rule of the simulation:

– *share*: We have

$$\frac{\Gamma, x : \tau' \vdash_{\mathcal{U}} E_{e1}[x] \sim e_s : \sigma \quad \Gamma \vdash_{\mathcal{U}} E_{e2}[e_e] \sim e'_s : \tau'}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = E_{e2}[e_e]\, \mathbf{in}\, E_{e1}[x] \sim (\lambda x{:}\tau'.e_s)e'_s : \sigma}$$

By the inductive hypothesis, we get that

$$\begin{array}{ll} \Gamma, x : \tau' \vdash e_s \mapsto^* E_{s1}[e_{s1}[\theta]] : \sigma & \Gamma \vdash e'_s \mapsto^* E_{s2}[e''_s[\theta]] : \tau' \\ \Gamma, x : \tau' \vdash E_{s1} \mathbin{@} e_{s1}[\theta] : \tau' \overset{\Gamma_1}{\Longrightarrow} \sigma & \Gamma \vdash E_{s2} \mathbin{@} e''_s[\theta] : \tau \overset{\Gamma_2}{\Longrightarrow} \tau' \\ \Gamma, x : \tau' \vdash_{\mathcal{U}} x \sim e_{s1} : \tau' & \Gamma, \Gamma_2 \vdash_{\mathcal{U}} e_e \sim e''_s : \tau \end{array}$$

It follows that e_{s1} must be x and $x \notin \mathrm{dom}(\theta)$, so because reduction is stable under substitution (Lemma 3) we reduce to

$$(\lambda x{:}\tau'.e_s)e'_s \mapsto_{\beta_{\leadsto}} e_s[e'_s/x] \mapsto^* E_{s1}[e'_s] \mapsto^* E_{s1}[E_{s2}[e''_s[\theta]]]$$

Any further reduction of e_e and e''_s preserve the relation by the *copy* rule, notably expanding the other instances of e'_s.

– *copy*: We have $E'_{e1}[\overline{x/y_i}] = E_{e1}$ and (without loss of generality) $x \notin \mathrm{FV}(E_{e1})$ such that

$$\frac{\Gamma, x : \tau', \overline{y_i : \tau'} \vdash_{\mathcal{U}} E_{e1}[x] \sim e_s : \sigma \quad \Gamma \vdash_{\mathcal{U}} E_{e2}[e_e] \sim e'_s : \tau' \quad \Gamma \vdash e'_{si} \mapsto^* e'_s : \tau'}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = E_{e2}[e_e]\, \mathbf{in}\, E_{e1}[x] \sim e_s[e'_{s0}/x, \overline{e'_{si}/y_i}] : \sigma}$$

By the inductive hypothesis, we get that

$$\begin{array}{ll} \Gamma, x : \tau', \overline{y_i : \tau'} \vdash e_s \mapsto^* E_{s1}[e_{s1}[\theta]] : \sigma & \Gamma \vdash e'_s \mapsto^* E_{s2}[e''_s[\theta]] : \tau' \\ \Gamma, x : \tau', \overline{y_i : \tau'} \vdash E_{s1} \mathbin{@} e_{s1}[\theta] : \tau' \overset{\Gamma_1}{\Longrightarrow} \sigma & \Gamma \vdash E_{s2} \mathbin{@} e''_s[\theta] : \tau \overset{\Gamma_2}{\Longrightarrow} \tau' \\ \Gamma, x : \tau', \overline{y_i : \tau'} \vdash_{\mathcal{U}} x \sim e_{s1} : \tau' & \Gamma, \Gamma_2 \vdash_{\mathcal{U}} e_e \sim e''_s : \tau \end{array}$$

It follows that e_{s1} must be x and $x \notin \mathrm{dom}(\theta)$, so we reduce to

$$e_s[e_{s0}/x, \overline{e'_{si}/y_i}] \mapsto^* E_{s1}[e_{s0}] \mapsto^* E_{s1}[e'_s] \mapsto^* E_{s1}[E_{s2}[e''_s[\theta]]]$$

Any further reduction of e_e and e''_s preserve the relation by the *copy* rule, further expanding the other instances of e'_s.

□

Lemma 12 (Finality Simulation). *If $\Gamma \vdash_{\mathcal{U}} Fin \sim e : \tau$ then $e \in Final$, and if $\Gamma \vdash_{\mathcal{U}} e \sim Fin : \tau$ then $e \in Final$. Likewise, if $\Gamma \vdash_{\mathcal{N}} Fin \sim e : \tau$ then $\Gamma \vdash e \mapsto^* Fin' : \tau$, and if $\Gamma \vdash_{\mathcal{N}} e \sim Fin : \tau$ then $e \in Final$.*

Proof. The first part about $\mathcal{U}$ follows immediately from Lemma 11 and the fact that final expressions are closed under substitution. The second part about $\mathcal{U}$ follows by induction on the simulation relation. The remaining statement about $\mathcal{N}$ follows by the reductions of the simulation, which are all instances of $\beta_{\rightsquigarrow}$. □

Lemma 13 (Forward Simulation). *If $\Gamma \vdash_{\mathcal{N}} e_1 \sim e_2 : \tau$ and $\Gamma \vdash e_1 \mapsto e'_1 : \tau$ by the call-by-need semantics then $\Gamma \vdash e_2 \mapsto^* e'_2 : \tau$ by the call-by-name semantics.*

Proof. First consider the cases where an operational step is applied directly to e_1 (*i.e.*, not *compat*). These cases are as follows:

- $\beta_{\rightsquigarrow}$, $\beta_{\forall}$, $\beta_{\{\}}$, and $\beta_{\mathtt{Int}}$ in call-by-need follow the rule of the same name in call-by-name.
- *rename* is either immediate (due to an application of *copy*) or follows by $\beta_{\rightsquigarrow}$ (due to an application of *share*).
- $name_{\mathtt{I\#}}$ and *name* follow from the *share* unwinding rules followed by a step of $unname_{\mathtt{I\#}}$ or *unname*, respectively.
- *look* and *comm* follow immediately when the **let** is simulated by *copy*, and follows by $\beta_{\rightsquigarrow}$ when simulated by *share* or *name*.

compat, follows by Lemma 11 and the fact that reductions are stable under substitution (Lemma 3). In more detail, suppose that

$$\frac{\Gamma \vdash E_e \mathbin{@} e_e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma \quad \Gamma, \Gamma' \vdash e_e \mapsto e'_e : \tau}{\Gamma \vdash E_e[e_e] \mapsto E_e[e'_e] : \sigma}$$

Then we know that, given any $\Gamma \vdash_{\mathcal{N}} E_e[e_e] \sim e_s : \sigma$, we have $\Gamma \vdash e_s \mapsto^* E_{s1}[e_{s1}[\theta]] \mapsto^* E_{s1}[e'_{s1}[\theta]] : \sigma$ such that $\Gamma \vdash_{\mathcal{N}} E_e[e'_e] \sim E_{s1}[e'_{s1}[\theta]]$. □

Lemma 14 (Context Rewinding). *If $\Gamma \vdash E_s \mathbin{@} e_s : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ in the call-by-name $\mathcal{IL}$ and $\Gamma \vdash_{\mathcal{U}} e'_e \sim E_s[e_s] : \sigma$, then*

Proof. By induction on the unwinding relation first, then on the decomposition of evaluation contexts.

- The cases for compatibility of unwinding with non-**let** syntax all follow by the inductive hypothesis.
- *share*: Where

$$\frac{\Gamma \vdash_{\mathcal{U}} e_e \sim e_s : \tau' \quad \Gamma, x : \tau' \vdash_{\mathcal{U}} = e'_e \sim e'_s : \sigma}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = e'_e \,\mathbf{in}\, e_e \sim (\lambda x{:}\tau' e_s)\ e'_s : \sigma}$$

The only possible decompositions of $(\lambda x{:}\tau' e_s)\ e'_s$ into an evaluation context is the empty context (for which the result follows immediately), or else $(\lambda x{:}\tau' e_s)\ E_s$ when $\Gamma \vdash \tau : \mathtt{TYPE}\ \rho\ \mathtt{Eval}^{\mathtt{U}}$ (for which the result follows from the inductive hypothesis).

- *copy*: Where

$$\frac{\Gamma \vdash_{\mathcal{U}} e'_e \sim e'_s : \tau' \quad \Gamma, \overline{y_i : \tau'} \vdash_{\mathcal{U}} e_e \sim e_s : \sigma \quad \Gamma \vdash e'_s\ \mathbf{val} \quad \Gamma \vdash e'_{si} \mapsto^* e'_s : \tau}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = e'_e\, \mathbf{in}\, e_e[\overline{x/y_i}] \sim e_s[\overline{e'_{si}/y_i}] : \sigma}\ copy$$

  There are two possibilities for the decomposition of $e_s[\overline{e'_{si}/y_i}]$ into an evaluation context surrounding a sub-expression (let $\theta = [e'_{s0}/y_0, \theta']$ and $\theta' = [e'_{s1}/y_1, \dots]$):
  - Commutation $e_s[\theta] = E_{s1}[\theta][e_{s1}[\theta]] = E_{s1}[e_{s1}][\theta]$ where the substitution does not interact with decomposition. In this case, we also have that $\Gamma, \overline{y_i : \tau'} \vdash E_{s1} @ e_{s1} : \tau \stackrel{\Gamma'}{\Longrightarrow} \sigma$ before substitution, and so the result follows from the inductive hypothesis.
  - Interference $e_s[\theta] = E_{s1}[\theta][E_{s2}[\theta][e_{s2}[\theta]]] = E_{s1}[y_0][E_{s2}[e_{s2}]/y_0, \theta']$ where the substitution lands in the middle of an evaluation context (without loss of generality, assume this is the only free appearance of y_0) and that substituted expression is further decomposed. In this case, the evaluation context is composed of $E_s = E_{s1}[E_{s2}]$ which includes part of an expression in the substitution θ. In other words, we have

  $$e_s[\theta] = E_{s1}[\theta'][e'_{s0}] = E_{s1}[\theta'][E_{s2}[e_{s2}]] \mapsto^* E_{s1}[\theta'][e'_s]$$

  where we know already that $\Gamma \vdash_{\mathcal{U}} e'_e \sim e'_s : \tau'$.

□

Lemma 15 (Backward Simulation). *If $\Gamma \vdash_{\mathcal{N}} e_1 \sim e_2 : \tau$ and $\Gamma \vdash e_2 \mapsto e'_2 : \tau$ by the call-by-name semantics then $\Gamma \vdash e_1 \mapsto^* e'_1 : \tau$ by the call-by-need semantics such that $\Gamma_{\mathcal{N}} \vdash e'_1 \sim e'_2 : \tau$.*

Proof. Note that any unnaming expansions are either $\beta_{\rightsquigarrow}$ redexes, or preserve the target of the evaluation context (in the case of a strict $\mathtt{Eval}^{\mathsf{U}}$ application), but these must be done first. Assuming there are additional reductions afterward, we can proceed by induction on the derivation the number of reduction steps in $\Gamma \vdash e_2 \mapsto e'_2 : \tau$ first and the underlying unwinding $\Gamma \vdash_{\mathcal{U}} e_1 \sim e_2 : \tau$ first then. In the cases for compatibility of unwinding, we either reduce a sub-expression, which follows by induction, or the top-level expression itself. The latter cases may be:

- $\beta_\forall$, $\beta_{\{\}}$, and $\beta_{\mathtt{Int}}$ all follow by the inductive hypothesis by applying the call-by-need rule of the same name, using Lemma 12 to ensure the body of the Λ-abstraction is still final in the case of $\beta_\forall$. Note that it is possible that a redex of the form $F[e]$ (where e **fin**) might have a **let** inserted in between the frame context and the final expression as $F[B[e]]$. In this case, an additional *comm* reduction is required.
- $\beta_{\rightsquigarrow}$: note that in call-by-need we have

  $$\Gamma \vdash (\lambda x{:}\tau.e_e)\ e'_e \mapsto_{name} \mathbf{let}\, x{:}\tau. = e'_e\, \mathbf{in} (\lambda x{:}\tau.e_e)\ x \mapsto_{\beta_{\rightsquigarrow}} \mathbf{let}\, x{:}\tau. = e'_e\, \mathbf{in}\, e_e : \sigma$$

  such that the *copy* rule relates this to the call-by-name $\beta_{\rightsquigarrow}$ reduct like so

  $$\frac{\Gamma \vdash_{\mathcal{U}} e'_e \sim e'_s : \tau \quad \Gamma, x{:}\tau \vdash_{\mathcal{U}} e_e \sim e_s : \sigma}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau. = e'_e\, \mathbf{in}\, e_e \sim e_s[e'_s/x]}\ copy$$

  This reduction is still possible if an additional **let**-binding is inserted in the middle of the $\beta_\forall$ redex, pushing the application to the argument a inside via *comm*.

Finally, we have the cases for unwinding a **let**-expression, which are as follows:

- *share*: Where

  $$\frac{\Gamma \vdash_{\mathcal{U}} e_e \sim e_s : \tau' \quad \Gamma, x : \tau' \vdash_{\mathcal{U}}= e'_e \sim e'_s : \sigma}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = e'_e\, \mathbf{in}\, e_e \sim (\lambda x{:}\tau' e_s)\ e'_s : \sigma}\ share$$

  The only possible decompositions of $(\lambda x{:}\tau' e_s)\ e'_s$ into an evaluation context are

  - the empty context, for which the only possible reduction is $\beta_{\rightsquigarrow}$, whose reduct is related to the original expression by *copy* instead, or
  - else $(\lambda x{:}\tau' e_s)\ E_s$ when $\Gamma \vdash \tau : \mathtt{TYPE}\ \rho\ \mathtt{Eval}^{\mathsf{U}}$, for which the result follows from the inductive hypothesis.

- *copy*: Where

$$\frac{\Gamma \vdash_{\mathcal{U}} e'_e \sim e'_s : \tau' \quad \Gamma, \overline{y_i : \tau'} \vdash_{\mathcal{U}} e_e \sim e_s : \sigma \quad \Gamma \vdash e'_s\ \mathbf{val} \quad \Gamma \vdash e'_{si} \mapsto^* e'_s : \tau}{\Gamma \vdash_{\mathcal{U}} \mathbf{let}\, x{:}\tau' = e'_e\ \mathbf{in}\, e_e[\overline{x/y_i}] \sim e_s[\overline{e'_{si}/y_i}] : \sigma}\ \mathit{copy}$$

  There are two possibilities for the decomposition of $e_s[\overline{e'_{si}/y_i}]$ into an evaluation context surrounding a sub-expression (let $\theta = [e'_{s0}/y_0, \theta']$ and $\theta' = [e'_{s1}/y_1, \dots]$):
  - Commutation $e_s[\theta] = E_{s1}[\theta][e_{s1}[\theta]] = E_{s1}[e_{s1}][\theta]$ where the substitution does not interact with decomposition. In this case, we also have that $\Gamma, \overline{y_i : \tau'} \vdash E_{s1} @ e_{s1} : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ before substitution, and so the result follows from the inductive hypothesis.
  - Interference $e_s[\theta] = E_{s1}[\theta][E_{s2}[\theta][e_{s2}[\theta]]] = E_{s1}[y_0][E_{s2}[e_{s2}]/y_0, \theta']$ where the substitution lands in the middle of an evaluation context (without loss of generality, assume this is the only free appearance of y_0) and that substituted expression is further decomposed. In this case, the evaluation context is composed of $E_s = E_{s1}[E_{s2}]$ which includes part of an expression in the substitution θ. In other words, we have

$$e_s[\theta] = E_{s1}[\theta'][e'_{s0}] = E_{s1}[\theta'][E_{s2}[e_{s2}]] \mapsto^* E_{s1}[\theta'][e'_s]$$

    where we know already that $\Gamma \vdash_{\mathcal{U}} e'_e \sim e'_s : \tau'$. Therefore, the reduction in call-by-name is either catching up to *copy*, or if there are any steps remaining afterward, the result follows by the inductive hypothesis. □

A.5 Correspondence to the equational theory

Here we show the correspondence between the equational theory (given in Section 4.9) and the call-by-name operational semantics for $\mathcal{IL}$ (given in Appendix A.2). Note that, crucially, the correspondence will hold specifically for expressions of final answer types τ_{fin}:

$$\tau_{\mathit{fin}}, \sigma_{\mathit{fin}} ::= \mathtt{Int\#} \mid \mathtt{Int}^{\gamma}$$

This restriction prevents the $\eta_{\rightsquigarrow}$ and $\eta_{\{\}}$ rules from exposing any underlying computation that needs to be evaluated, and thus, they are unnecessary for evaluating the final answer.

Theorem 5. *If* $\Gamma \vdash e \mapsto^* e' : \tau$ *then* $\Gamma \vdash e = e' : \tau$.

Proof. Each call-by-name operational step is an instance of an equational axiom of $\mathcal{IL}$. □

For the other direction, we can show via confluence and standardization. First, let reduction relation

$$\Gamma \vdash e \to e' : \tau$$

be defined as the generalization of the call-by-name operational semantics in Appendix A.2 so that compatibility (*compat*) applies to *any* context, as well as the generalization of the $\beta_\forall$ rule to the left-to-right reading of $\beta_\forall$ rule in Fig. 4. In other words, $\beta_\forall$ applies to any polymorphic instantiation $(\Lambda\chi.e)\ \phi$, not just ones where the body e is final. Note that the reflexive-transitive-symmetric closure of this reduction theory the same as the equational theory presented in Fig. 4, even though the $\beta_{\rightsquigarrow}$ rule only applies to *values* of kind $\mathtt{Eval}^{\mathsf{U}}$ rather than *answers*. That's because all answers evaluate to values, anyway.

Lemma 16 (Answer Value). *If* $\Gamma \vdash A : \tau$ *then* $\Gamma \vdash A \mapsto V : \tau$.

Theorem 6 (Confluence). *If* $\Gamma \vdash e \rightarrow^* e_1 : \tau$ *and* $\Gamma \vdash e \rightarrow^* e_2 : \tau$ *then* $\Gamma \vdash e_1 \rightarrow^* e' : \tau$ *and* $\Gamma \vdash e_2 \rightarrow^* e' : \tau$ *for some* e'.

Proof. The only critical pairs in the reduction rules are between matching β and η rules, which join together immediately (in zero steps). As such, the reduction theory forms a combinatorial, orthogonal rewrite system. □

Corollary 3 (Church-Rosser). $\Gamma \vdash e = e' : \tau$ *if and only if* $\Gamma \vdash e \rightarrow^* e'' \leftarrow^* e' : \tau$.

Proof. Follows from Theorem 6. □

Definition 2 (Internal Reduction). An *internal reduction*, written $\Gamma \vdash e \rightharpoonup e' : \tau$, is any reduction $\Gamma \vdash e \rightarrow e' : \tau$ such that $\Gamma \vdash e \not\mapsto e' : \tau$.

Internal reductions (that is, ones that differ from the next operational step) are useful because they are never needed to convert expressions into final ones.

Lemma 17 (Final Stability).

(1) If $\Gamma \vdash e \rightharpoonup e' : \tau$, *then* $e \in \mathit{Final}$ *if and only if* $e' \in \mathit{Final}$, *and* $\Gamma \vdash e$ **val** *if and only if* $\Gamma \vdash e'$ **val**.
(2) If $\Gamma \vdash e \rightarrow e' : \tau_{\mathit{fin}}$, *then* $e' \in \mathit{Final}$ *whenever* $e \in \mathit{Final}$, *and* $\Gamma \vdash e'$ **val** *whenever* $\Gamma \vdash e$ **val**.

Proof. By induction on the syntax of final expressions and derivation of **val**, then cases on the possible internal reductions. Note that the restriction on the type in the second part prevents an $\eta_{\rightsquigarrow}$ or $\eta_{\{\}}$ reduction from exposing a non-final expression. □

Definition 3 (Parallel Reduction). The *parallel reduction relation*, written $\Gamma \vdash e \mapsto\!\!\!\!\Rightarrow e' : \tau$, is given by the restriction on $\Gamma \vdash e \rightarrow^* e' : \tau$ to only non-overlapping redexes that are all present originally in e. The *internal parallel reduction relation*, written $\Gamma \vdash e \Rightarrow e' : \tau$, is the restriction of parallel reduction to only internal reductions.

Parallel reduction is interesting because, unlike an ordinary single reduction step, it commutes with substitution in one parallel step:

Lemma 18 (Parallel Substitution).
If $\Gamma, x : \tau \vdash e : \sigma$ *and* $\Gamma \vdash e' \mapsto\!\!\!\!\Rightarrow e'' : \tau$ *then* $\Gamma \vdash e[e'/x] \mapsto\!\!\!\!\Rightarrow e[e''/x] : \sigma$.

Proof. By induction on the derivation of $\Gamma, x : \tau \vdash e : \sigma$. □

Lemma 19 (Internal Decomposition). *If* $\Gamma \vdash e \Rightarrow E_2[e_2] : \sigma_{\mathit{fin}}$ *and* $\Gamma \vdash E_2 \mathbin{@} e_2 : \tau \xRightarrow{\Gamma'} \sigma_{\mathit{fin}}$ *then there is an* E_1 *and* $\theta = [\overline{\phi/\chi}]$ *and* τ' *such that*

- $e =_\alpha E_1[e_1]$ *and* $\Gamma \vdash E_1 \mathbin{@} e_1 : \tau' \xRightarrow{\Gamma', \overline{\chi}} \sigma_{\mathit{fin}}$,
- $e_2 =_\alpha e_2[\theta]$ *and* $\Gamma, \Gamma', \chi \vdash e_1 \Rightarrow e_2 : \tau$, *and*
- $\Gamma \vdash E[e_3] \Rightarrow E'[e_3[\theta]] : \sigma_{\mathit{fin}}$ *for any* $\Gamma, \Gamma', \overline{\chi} \vdash e_3 : \tau$.

Proof. By induction on the syntax of evaluation contexts, and inversion on the possible internal reductions. The substitutions θ are possible due to internal applications of $\beta_\forall$ to an abstraction $\Lambda\chi.e$ with e not final. Note that the restriction on the final answer type required to prevent an internal $\eta_{\rightsquigarrow}$ or $\eta_{\{\}}$ reduction from exposing a deeper evaluation context inside the body of a Λ-abstraction or closure. As such, each application of these two η rules must either occur outside the eye of the evaluation context, or inside a $\beta_{\rightsquigarrow}$ or $\beta_{\{\}}$ redex. In the latter case, such η reductions mimic β operational steps, therefore making them non-internal and ruling them out. □

Lemma 20 (Standard Preponement). *If* $\Gamma \vdash e \Rightarrow e_1 \mapsto^* e' : \tau_{\mathit{fin}}$ *then* $\Gamma \vdash e \mapsto^* e_2 \mapsto\!\!\!\!\Rightarrow e' : \tau_{\mathit{fin}}$.

Proof. First, we show the case for a single step: If $\Gamma \vdash e \Rightarrow e_1 \mapsto e' : \tau$ then $\Gamma \vdash e \mapsto^* e_2 \Rrightarrow e' : \tau$ for some e_2. The cases for applying an operational step directly to an expression are as follow from Lemma 18. For example, the $\beta_{\leadsto}$ step is as follows: assume that

$$\Gamma \vdash (\lambda x{:}\tau.e_1)\ e_2 \Rightarrow (\lambda x{:}\tau.e_1')\ e_2' \mapsto_{\beta_{\leadsto}} e_1'[e_2'/x] : \sigma$$

because $\Gamma, x : \tau \vdash e_1 \Rrightarrow e_1' : \sigma$ and $\Gamma \vdash e_2 \Rrightarrow e_2' : \tau$ and $\Gamma \vdash e_2'$ **val**. Note that $\Gamma \vdash e_2$ **val** due to Lemma 17, so we can do the $\beta_{\leadsto}$ step first, like so:

$$\Gamma \vdash (\lambda x{:}\tau.e_1)\ e_2 \mapsto_{\beta_{\leadsto}} e_1[e_2/x] \Rrightarrow e_1'[e_2'/x]$$

The other cases are similar. Compatibility of an operational steps inside an evaluation context follows from induction on the derivation and Lemma 19. Notably, if we have

$$\frac{\Gamma \vdash E_2 \mathbin{@} e_2 : \sigma \xRightarrow{\Gamma'} \tau_{fin} \qquad \Gamma, \Gamma' \vdash e_2 \mapsto e_2' : \sigma}{\Gamma \vdash e \Rightarrow E_2[e_2] \mapsto E_2[e_2'] : \tau_{fin}}$$

Lemmas 3 and 19 ensure there is an E_1, e_1, and $\theta = [\overline{\phi/\chi}]$ such that the beginning $e =_\alpha E_1[e_1]$, the reduced $e_2 =_\alpha e_3[\theta]$, and:

$$\Gamma, \Gamma', \overline{\chi} \vdash e_1 \Rightarrow e_3 \mapsto e_3' : \tau_{fin} \qquad\qquad \Gamma, \Gamma \vdash e_1[\theta] \Rightarrow e_2 \mapsto e_3'[\theta] : \tau_{fin}$$

meaning that $e_2' =_\alpha e_3'[\theta]$ by Corollary 1. Therefore, from the inductive hypothesis and compatibility:

$$\Gamma, \Gamma \vdash e_1 \mapsto^* e_4 \Rrightarrow e_3' : \tau_{fin} \qquad\qquad \Gamma \vdash E_1[e_1] \mapsto^* E_1[e_4] \Rrightarrow E_2[e_3'[\theta]] =_\alpha E_2[e_2'] : \tau_{fin}$$

Finally, the case for multiple steps of the operational semantics follows from the single-step case by induction on the transitive closure of the stepping relation. □

Lemma 21 (Internal Postponement). *If $\Gamma \vdash e \rightharpoonup^* e_1 \mapsto^* e' : \tau_{fin}$ then $\Gamma \vdash e \mapsto^* e_2 \rightharpoonup^* e' : \tau_{fin}$.*

Proof. Note that $\rightharpoonup^*$ and $\Rrightarrow^*$ are the same relation, so the theorem is equivalent to: If $\Gamma \vdash e \Rrightarrow^* e_1 \mapsto^* e' : \tau_{fin}$ then $\Gamma \vdash e \mapsto^* e_2 \Rrightarrow^* e' : \tau_{fin}$. This follows from Lemma 20 by induction on the reflexive-transitive closure of $\Rrightarrow^*$. □

Lemma 22 (Standard Order). *If $\Gamma \vdash e \rightarrow^* e' : \tau_{fin}$ then $\Gamma \vdash e \mapsto^* e'' \rightharpoonup^* e' : \tau_{fin}$*

Proof. Every reduction sequence corresponds to alternations between operational steps and internal reductions:

$$e_1 \rightarrow^* e_n \text{ iff } e_1 \rightharpoonup^* e_1 \mapsto^* \dots \rightharpoonup^* e_{n-1} \mapsto^* e_n$$

Therefore, by induction on the number of these alternations, we can reorder all the operational steps to come first with Lemma 21. □

Theorem 7 (Standardization). *If $\Gamma \vdash e \rightarrow^* Fin : \tau_{fin}$ then there is a $\Gamma \vdash Fin' \rightarrow^* Fin : \tau_{fin}$ such that $\Gamma \vdash e \mapsto^* Fin' : \tau_{fin}$.*

Proof. From Lemma 22, we know that $\Gamma \vdash e \mapsto^* e' \rightharpoonup^* Fin : \tau$ for some e', and from Lemma 17 e' must be final. □

Corollary 4. *If $\Gamma \vdash e = Fin : \tau_{fin}$ then $\Gamma \vdash e \mapsto^* Fin' : \tau_{fin}$ such that $\Gamma \vdash Fin' = Fin : \tau_{fin}$.*

Proof. From Corollary 3 we know that $\Gamma \vdash e \rightarrow^* Fin' \leftarrow^* Fin : \tau_{fin}$, and from Theorem 7 we know that $\Gamma \vdash e \mapsto^* Fin'' \rightarrow^* Fin' \leftarrow^* Fin : \tau_{fin}$ □

B CORRECTNESS OF $\mathcal{IL}$-TO-$\mathcal{ML}$ COMPILATION

In this section, assume the use of black holes to mask forced thunks:

$$(\mathit{Force}) \qquad \langle x_{\mathtt{PtrR}} \mid K \mid [x := \mathbf{memo}\ e]H \rangle \mapsto \langle e \mid \mathbf{set}\ x; K \mid [x := \bullet]H \rangle$$
$$(\mathit{Memo}) \qquad \langle R \mid \mathbf{set}\ x; K \mid [x := \bullet]H \rangle \mapsto \langle R \mid K \mid [x := R]H \rangle$$

B.1 Well-typed Expressions Can be Compiled

While we might only be interested in applying the compiler to whole, closed programs, we still need to be able to handle fragments of that program during compilation. In general, we must consider compiling open expressions of different types which may have free variables in them. To do so, we need to calculate the primitive representation of function arguments a, which is defined as:

$$x_\pi \overset{rep}{\rightsquigarrow} \pi \qquad\qquad i \overset{rep}{\rightsquigarrow} \texttt{IntR} \qquad\qquad \texttt{error} \overset{rep}{\rightsquigarrow} \texttt{PtrR}$$

These free variables are tracked in both a typing context Γ (used for type checking sub-expressions during compilation) as well as a renaming environment θ (used for replacing $\mathcal{IL}$ variables with $\mathcal{ML}$ ones). We need to check these two forms of environments correspond to one another, and also that each $\mathcal{ML}$ variable has a known representation matching the type in Γ, as follows:

$$\Gamma \vdash \theta \ \textbf{mono-rep} \text{ iff for any } x{:}\tau \in \Gamma, \Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi \text{ and } \theta(x) \overset{rep}{\rightsquigarrow} \pi$$

Note that the main compilation of expressions, $\mathcal{C}[\![e]\!]^{\Gamma}_{\theta}(\overline{a})$ generates code which evaluates e. In other words, this operation is always *strict* in e. As such, it does not matter if e is lifted or unlifted: evaluation of e is being forced either way. In general, compilation needs to know about the convention of e—if it is just being evaluated or called with a list of parameters—in order to generate the appropriate $\mathcal{ML}$ code. However, it does *not* need to know its levity, since the code that is generated will only be run when the result is needed anyway. To state this assumption formally, we need a more relaxed notion of calculating the calling convention of a type even if the levity is unknown—in other words, a levity polymorphic version of $\tau \overset{conv}{\rightsquigarrow} \nu$—as defined by the following rules for $\Gamma \vdash \tau \overset{conv-lp}{\rightsquigarrow} \eta$:

$$\frac{\Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta}{\Gamma \vdash \tau \overset{conv-lp}{\rightsquigarrow} \eta} \qquad \frac{\Gamma \vdash \tau : \texttt{TYPE}\ \rho\ \texttt{Eval}^{l}}{\Gamma \vdash \tau \overset{conv-lp}{\rightsquigarrow} \texttt{Eval}^{l}}$$

With the notion of a levity-polymorphic, but otherwise statically known, calling convention, we can state the invariants for when static compilation is defined:

Lemma 23 (Open Compilation). *$\mathcal{E}_\nu[\![e]\!]^{\Gamma}_{\theta}$ is defined if:*

(1) $\Gamma \vdash e : \tau$ is derivable for some type τ,
(2) $\Gamma \vdash \tau \overset{conv-lp}{\rightsquigarrow} \nu$ for some ν, and
*(3) $\Gamma \vdash \theta$ **mono-rep**.*

Proof. By induction on the given typing derivation of $\Gamma \vdash e : \tau$. Note the following invariants to ensure that $\mathcal{C}[\![e]\!]^{\Gamma}_{\theta}(\overline{a})$ is defined:

(1) $\Gamma \vdash e : \tau$ is derivable for some type τ,
(2) $\Gamma \vdash \tau \overset{conv-lp}{\rightsquigarrow} \nu$ for some ν, such that $|\overline{a}| = |arity(\nu)|$, and $a_i \overset{rep}{\rightsquigarrow} \pi_i$ for each $\pi_i \in arity(\nu)$, and
(3) $\Gamma \vdash \theta$ **mono-rep**.

Furthermore, the only invariants required for $\mathcal{A}[\![A]\!]^{\Gamma}_{\theta}$ to be defined are that (1) $\Gamma \vdash e : \tau$ is derivable for some type τ, (2) $\Gamma \vdash \theta$ **mono-rep**. The side conditions in rules requiring $\Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi$ and $\Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta$ are ensured by the following facts about the monomorphism restrictions:

(1) If $\vdash \rho$ **rep**, then $\rho = \pi$ for some π.
(2) If $\vdash \gamma$ **lev**, then $\gamma = \psi$ for some ψ.
(3) If $\vdash \nu$ **conv**, then either $\nu = \eta$ for some η. □

B.2 An Expression-based Operational Semantics for $\mathcal{ML}$

$$
\begin{array}{rrcl}
\mathit{Final} \ni & \mathit{Fin} & ::= & H[V] \mid E[\mathtt{error}(n)] \\
\mathit{EvalCxt} \ni & E & ::= & \square \mid F[E] \mid B[E] \\
\mathit{ClosCxt} \ni & H & ::= & \square \mid B[H] \\
\mathit{FrameCxt} \ni & F & ::= & \mathsf{App}\ \square(\overline{a}) \mid \mathbf{case}\ \square\ \mathbf{of}\ \mathtt{I\#}(x_{\mathtt{IntR}}) \to e \mid \mathbf{let}\ x_\pi^{\mathsf{U}} = \square\ \mathbf{in}\ e \mid \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = \square\ \mathbf{in}\ E[x_{\mathtt{PtrR}}] \\
\mathit{BindCxt} \ni & B & ::= & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ \square \mid \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = e\ \mathbf{in}\ \square
\end{array}
$$

Main reductions:

$$
\begin{array}{rrcll}
(\mathit{Call}) & (\lambda(\overline{x_\pi}).e)\ (\overline{a}) & \mapsto & e[\overline{a/x_\pi}] & \\
(\mathit{Apply}) & \mathsf{App}\ (\mathtt{Clos}^n\ W)(\overline{a}) & \mapsto & W(\overline{a}) & (\text{if } |\overline{a}| = n) \\
(\mathit{Move}) & \mathbf{let}\ x_\pi^{\mathsf{U}} = c\ \mathbf{in}\ e & \mapsto & e[c/x_\pi] & \\
(\mathit{Unbox}) & \mathbf{case}\ \mathtt{I\#}(a)\ \mathbf{of}\ \mathtt{I\#}(x_{\mathtt{IntR}}) \to e & \mapsto & e[a/x_{\mathtt{IntR}}] &
\end{array}
$$

Pointer lookup and memoization (assume E does not bind $x_{\mathtt{PtrR}}$):

$$
\begin{array}{rrcl}
(\mathit{Fun}) & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ E[x_{\mathtt{PtrR}}(\overline{a})] & \mapsto & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ E[R(\overline{a})] \\
(\mathit{Look}) & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ E[x_{\mathtt{PtrR}}] & \mapsto & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ E[R] \\
(\mathit{Memo}) & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = R\ \mathbf{in}\ E[x_{\mathtt{PtrR}}] & \mapsto & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ E[R]
\end{array}
$$

Percolating frame contexts out of binding contexts (assume $x \notin \mathrm{FV}(F)$):

$$
\begin{array}{rrcl}
(\mathit{LAlloc}) & F[\mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = e\ \mathbf{in}\ \mathit{Fin}] & \mapsto & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = e\ \mathbf{in}\ F[\mathit{Fin}] \\
(\mathit{SAlloc}) & F[\mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ \mathit{Fin}] & \mapsto & \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ F[\mathit{Fin}]
\end{array}
$$

B.3 Bisimulation between $\mathcal{ML}$'s operational semantics and abstract machine

Translating evaluation contexts to stacks and heaps:

$$
\begin{array}{rclcrcl}
\mathcal{K}[\![\square]\!] & \triangleq & \varepsilon & \qquad & \mathcal{H}[\![\square]\!] & \triangleq & \varepsilon \\
\mathcal{K}[\![F[E]]\!] & \triangleq & \mathcal{K}[\![E]\!] \circ \mathcal{K}[\![F]\!] & & \mathcal{H}[\![F[E]]\!] & \triangleq & \mathcal{H}[\![E]\!] \circ \mathcal{H}[\![F]\!] \\
\mathcal{K}[\![B[E]]\!] & \triangleq & \mathcal{K}[\![E]\!] & & \mathcal{H}[\![B[E]]\!] & \triangleq & \mathcal{H}[\![B]\!] \circ \mathcal{H}[\![E]\!]
\end{array}
$$

$$
\begin{array}{rclcrcl}
\mathcal{K}[\![\mathsf{App}\ \square(\overline{a})]\!] & \triangleq & \mathsf{App}(\overline{a}); \varepsilon & \qquad & \mathcal{H}[\![\mathsf{App}\ \square(\overline{a})]\!] & \triangleq & \varepsilon \\
\mathcal{K}[\![\mathbf{case}\ \square\ \mathbf{of}\ \mathtt{I\#}(x_{\mathtt{IntR}}){\to}e]\!] & \triangleq & \mathbf{case}\ \mathtt{I\#}(x_{\mathtt{IntR}}){\to}e; \varepsilon & & \mathcal{H}[\![\mathbf{case}\ \square\ \mathbf{of}\ \mathtt{I\#}(x_{\mathtt{IntR}}){\to}e]\!] & \triangleq & \varepsilon \\
\mathcal{K}[\![\mathbf{let}\ x_\pi^{\mathsf{U}} = \square\ \mathbf{in}\ e]\!] & \triangleq & \mathbf{let}\ x_\pi\ \mathbf{in}\ e; \varepsilon & & \mathcal{K}[\![\mathbf{let}\ x_\pi^{\mathsf{U}} = \square\ \mathbf{in}\ e]\!] & \triangleq & \varepsilon \\
\mathcal{K}[\![\mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = \square\ \mathbf{in}\ E[x_{\mathtt{PtrR}}]]\!] & \triangleq & \mathbf{set}\ x; \mathcal{K}[\![E]\!] & & \mathcal{H}[\![\mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = \square\ \mathbf{in}\ E[x_{\mathtt{PtrR}}]]\!] & \triangleq & \mathcal{H}[\![E]\!][x := \bullet]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{H}[\![\mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{U}} = R\ \mathbf{in}\ \square]\!] & \triangleq & [x := R] \\
\mathcal{H}[\![\mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = e\ \mathbf{in}\ \square]\!] & \triangleq & [x := \mathbf{memo}\ e]
\end{array}
$$

Definition 4 (Refocusing)**.** The *refocusing* steps of the abstract machine are *PshApp*, *PshCase*, *PshLet*, *LAlloc*, *SAlloc*, *Force*, and *Error*. We write $m \mapsto_F m'$ for a transition by one of the refocusing steps, and $m \mapsto_R m'$ for a transition by a non-refocusing reduction step.

Lemma 24. $\langle E[e] \mid K \mid H \rangle \mapsto_F^* \langle e \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{H}[\![E]\!] \circ H \rangle$

Proof. By induction on the syntax of E. The most interesting case is for an evaluation context of the form $\mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = E_2\ \mathbf{in}\ E_1[x_{\mathtt{PtrR}}]$, which proceeds as follows:

$$
\begin{array}{rl}
\langle \mathbf{let}\ x_{\mathtt{PtrR}}^{\mathsf{L}} = E_2[e]\ \mathbf{in}\ E_1[x_{\mathtt{PtrR}}] \mid K \mid H \rangle & \mapsto_{\mathit{Alloc}} \langle E_1[x_{\mathtt{PtrR}}] \mid K \mid [x := \mathbf{memo}\ E_2[e]]H \rangle \\
& \mapsto_{\mathit{IH}} \langle x_{\mathtt{PtrR}} \mid \mathcal{K}[\![E_1]\!] \circ K \mid \mathcal{H}[\![E_1]\!] \circ [x := \mathbf{memo}\ E_2[e]]H \rangle \\
& \mapsto_{\mathit{Force}} \langle E_2[e] \mid \mathcal{K}[\![E_1]\!] \circ K \mid \mathcal{H}[\![E_1]\!] \circ [x := \bullet]H \rangle \\
& \mapsto_{\mathit{IH}} \langle e \mid \mathcal{K}[\![E_2]\!] \circ \mathcal{K}[\![E_1]\!] \circ K \mid \mathcal{H}[\![E_2]\!] \circ \mathcal{H}[\![E_1]\!] \circ [x := \bullet]H \rangle
\end{array}
$$

And note that the corresponding stack is $\mathcal{K}[\![\mathbf{let}\, x^{\mathsf{L}}_{\mathsf{PtrR}} = E_2 \,\mathbf{in}\, E_1[x_{\mathsf{PtrR}}]]\!] = \mathcal{K}[\![E_2]\!] \circ \mathcal{K}[\![E_1]\!]$ and heap is $\mathcal{H}[\![\mathbf{let}\, x^{\mathsf{L}}_{\mathsf{PtrR}} = E_2 \,\mathbf{in}\, E_1[x_{\mathsf{PtrR}}]]\!] = \mathcal{H}[\![E_2]\!] \circ \mathcal{H}[\![E_1]\!] \circ [x := \bullet]$ □

Corollary 5 (Finality Preservation)**.** *If $Fin \sim m$ then $m \mapsto^*_F Fin'$. If $e \sim Fin$, then $e \in Final$.*

Proof. By definition of final expressions and machine states in $\mathcal{ML}$, from Lemma 24, taking a final *Error* step in the erroneous case $E[\mathtt{error}(n)]$. □

Definition 5 (Machine Simulation)**.** The simulation relation between $\mathcal{ML}$ expressions and machine states is

$$e \sim m \text{ iff } \langle e \mid \varepsilon \mid \varepsilon \rangle \mapsto^*_F m$$

Lemma 25 (Forward Simulation)**.** *(1) If $e \mapsto e'$ by LAlloc, or SAlloc and $e \sim m$ then $e' \sim m$.*
(2) If $e \mapsto e'$ by any other step and $e \sim m$ then $m \mapsto^*_F m' \mapsto_R m''$ and $e' \sim m''$.
Therefore, if $e \mapsto^* e'$ and $e \sim m$ then $m \mapsto^* m'$ and $e' \sim m'$ for some m'.

Proof. First, consider the cases where $e \mapsto e'$ by applying a single step of the reduction rules directly. Each of the rules in the operational semantics corresponds to the machine rule of the same name.

- *LAlloc* is, due Lemma 24 and the fact that the heap is unordered:

$$\begin{aligned}
\langle F[\mathbf{let}\, x^{\mathsf{L}}_{\mathsf{PtrR}} = e \,\mathbf{in}\, Fin] \mid \varepsilon \mid \varepsilon \rangle &\mapsto^*_F \langle \mathbf{let}\, x^{\mathsf{L}}_{\mathsf{PtrR}} = e \,\mathbf{in}\, Fin \mid \mathcal{K}[\![F]\!] \mid \mathcal{H}[\![F]\!] \rangle \\
&\mapsto_{LAlloc} \langle Fin \mid \mathcal{K}[\![F]\!] \mid [x := \mathbf{memo}\, e]\mathcal{H}[\![F]\!] \rangle \\
&\leftarrow\!\!\!\mid^*_F \langle F[Fin] \mid \varepsilon \mid [x := \mathbf{memo}\, e] \rangle \\
&\leftarrow\!\!\!\mid_F \langle \mathbf{let}\, x^{\mathsf{L}}_{\mathsf{PtrR}} = e \,\mathbf{in}\, F[Fin] \mid \varepsilon \mid \varepsilon \rangle
\end{aligned}$$

- *SAlloc* is similar to *LAlloc*.
- *Call* is $\langle (\lambda(\overline{x_\pi}).e)(\overline{a}) \mid \varepsilon \mid \varepsilon \rangle \mapsto_{Call} \langle e[\overline{a/x_\pi}] \mid \varepsilon \mid \varepsilon \rangle$.
- *Move* is similar to *Call*.
- *Apply* is $\langle \mathtt{App}\ (\mathtt{Clos}^n\, W)(\overline{a}) \mid \varepsilon \mid \varepsilon \rangle \mapsto_{PshApp} \langle \mathtt{Clos}^n\, W \mid \mathtt{App}(\overline{a}); \varepsilon \mid \varepsilon \rangle \mapsto_{Apply} \langle W(\overline{a}) \mid \varepsilon \mid \varepsilon \rangle$.
- *Unbox* is similar to *Apply*
- *Look* is

$$\begin{aligned}
\langle \mathbf{let}\, x^{\mathsf{U}}_{\mathsf{PtrR}} = R \,\mathbf{in}\, E[x_{\mathsf{PtrR}}] \mid K \mid H \rangle &\mapsto^*_F \langle E[x_{\mathsf{PtrR}}] \mid K \mid [x := R]H \rangle \\
&\mapsto^*_F \langle x_{\mathsf{PtrR}} \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{K}[\![E]\!] \circ [x := R]H \rangle \\
&\mapsto_{Look} \langle R \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{K}[\![E]\!] \circ [x := R]H \rangle \\
&\leftarrow\!\!\!\mid^*_F \langle E[R] \mid K \mid [x := R]H \rangle \\
&\leftarrow\!\!\!\mid^*_F \langle \mathbf{let}\, x^{\mathsf{U}}_{\mathsf{PtrR}} = R \,\mathbf{in}\, E[R] \mid K \mid H \rangle
\end{aligned}$$

- *Fun* is similar.
- *Memo* is

$$\begin{aligned}
\langle \mathbf{let}\, x^{\mathsf{L}}_{\mathsf{PtrR}} = R \,\mathbf{in}\, E[x_{\mathsf{PtrR}}] \mid K \mid H \rangle &\mapsto_F \langle E[x_{\mathsf{PtrR}}] \mid K \mid [x := \mathbf{memo}\, R]H \rangle \\
&\mapsto^*_F \langle x_{\mathsf{PtrR}} \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{H}[\![E]\!] \circ [x := \mathbf{memo}\, R]H \rangle \\
&\mapsto_F \langle R \mid \mathbf{set}\, x; \mathcal{K}[\![E]\!] \circ K \mid \mathcal{H}[\![E]\!] \circ [x := \bullet]H \rangle \\
&\mapsto_{Memo} \langle R \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{H}[\![E]\!] \circ [x := R]H \rangle \\
&\leftarrow\!\!\!\mid_{Look} \langle x_{\mathsf{PtrR}} \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{H}[\![E]\!] \circ [x := R]H \rangle \\
&\leftarrow\!\!\!\mid^*_F \langle E[x_{\mathsf{PtrR}}] \mid K \mid [x := R]H \rangle \\
&\leftarrow\!\!\!\mid^*_F \langle \mathbf{let}\, x^{\mathsf{U}}_{\mathsf{PtrR}} = R \,\mathbf{in}\, E[x_{\mathsf{PtrR}}] \mid K \mid H \rangle
\end{aligned}$$

Now, assume that $e \mapsto e'$ by applying one of the reduction rules as in the above steps, by Lemma 24 and due to the fact that refocusing steps are deterministic, reduction within an evaluation context $E[e] \mapsto E[e']$ proceeds like so:

$$\begin{aligned} \langle E[e] \mid K \mid H \rangle \mapsto_F^* m \mapsto_F^* \langle e \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{H}[\![E]\!] \circ H \rangle \\ \mapsto^* m' \leftarrow\!\shortmid_F^* \langle e' \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{H}[\![E]\!] \circ H \rangle \,{}_F^* \!\leftarrow\!\shortmid \langle E[e'] \mid K \mid H \rangle \end{aligned}$$

so that $E[e'] \sim \langle e' \mid \mathcal{K}[\![E]\!] \circ K \mid \mathcal{H}[\![E]\!] \circ H \rangle$. Finally, multiple reductions $e \mapsto^* e'$ follows from transitivity by induction on the number of steps. □

Lemma 26 (Backward Simulation). *If $m \mapsto^* m'$ and $e \sim m$ then $e' \sim m'$ and $e \mapsto^* e'$ for some e'.*

Proof. Note that if $m \mapsto_F^* m'$ then $e \sim m'$ directly. Otherwise, if $m \mapsto_R m'$ then $e \mapsto_{LAlloc,SAlloc}^* e'' \mapsto e'$ and by forward simulation Lemma 25 and determinism of the abstract machine, we know that $e' \sim m'$. □

B.4 Bisimulation between $\mathcal{IL}$ and $\mathcal{ML}$

Here, we relate the environment-based call-by-need operational semantics for $\mathcal{IL}$ with the operational semantics for $\mathcal{ML}$, so in the following, assume those definitions given in Appendix A.3. The simulation relationship between the two languages is given by compilation. Note that the compilation of **let**-expressions can be derived from the encoding of **let**s as applied λ-abstractions.

Definition 6 (Compilation Simulation). The simulation relation between open, typed $\mathcal{IL}$ expressions $\Gamma \vdash e : \tau : \mathtt{TYPE}\ \rho\ \nu$ and $\mathcal{ML}$ expressions is defined as

$$\Gamma \vdash_C e \sim e' : \tau \text{ iff } e' \mapsto_{Unlift}^* C[\![e]\!]_\theta^\Gamma(\overline{a})$$

for some θ and $\overline{a}$ such that $\Gamma \vdash \theta$ **mono-rep** and $\overline{a} \overset{rep}{\rightsquigarrow} arity(\nu)$, where the $Unlift$ reduction is:

$$(Unlift) \qquad\qquad \mathbf{let}\, x^{\mathsf{L}}_{\mathtt{PtrR}} = R\, \mathbf{in}\, e \mapsto \mathbf{let}\, x^{\mathsf{U}}_{\mathtt{PtrR}} = R\, \mathbf{in}\, e$$

For the purpose of this simulation, we treat an *Apply* followed by a *Call* as a single step in $\mathcal{ML}$.

Lemma 27 (Answer Stability). *$\mathcal{E}_\nu[\![A]\!]_\theta^\Gamma \triangleq \mathcal{A}[\![A]\!]_\theta^\Gamma$ and $C[\![A']\!]_\theta^\Gamma(\varepsilon) \triangleq \mathcal{A}[\![A']\!]_\theta^\Gamma$ when defined and when A' is not a variable.*

Proof. By induction on the syntax of A. □

Lemma 28 (Finality Preservation). *If $\Gamma \vdash Fin$ **fin** and $\Gamma \vdash_C Fin \sim e : \tau$ then $e \in Final$. If $\Gamma \vdash_C e \sim Fin : \tau$, then $e \in Final$.*

Proof. The first part follows by induction on the syntax of final $\mathcal{IL}$ expressions. Of note, in the base cases are where Fin is a reference $\mathtt{I\#}^\gamma$ arg or $\mathtt{Clos}^\gamma\, e$ (which have a convention of $\mathtt{Eval}^\gamma$ and cannot be applied to any arguments) or a constant $c\,\overline{\phi}$ (which could have any calling convention but cannot reduce further, even when applied). Dually, the only $\mathcal{IL}$ expressions that can compile to final $\mathcal{ML}$ machine states are final $\mathcal{IL}$ expressions. □

Translation of $\mathcal{IL}$ evaluation contexts (surrounding an expression of type τ) to $\mathcal{ML}$ evaluation contexts is written as $C_\tau[\![E]\!]_\theta^\Gamma(\overline{a})$. The definition for most cases, where strictness is obvious from the syntax, are defined as follows:

$$
\begin{aligned}
C_\tau[\![\square]\!]^\Gamma_\theta(\overline{a}) &\triangleq \square\,\overline{a}\\
C_\tau[\![E\ a']\!]^\Gamma_\theta(\overline{a}) &\triangleq C_\tau[\![E]\!]^\Gamma_\theta(\mathcal{A}[\![a']\!]^\Gamma_\theta, \overline{a})\\
C_\tau[\![\mathbf{case}\,E\,\mathbf{of}\ \mathtt{I\#}\ x \longrightarrow e]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathbf{case}\,C_\tau[\![E]\!]^\Gamma_\theta(\varepsilon)\ \mathbf{of}\ \mathtt{I\#}\ x \longrightarrow C[\![e]\!]^\Gamma_\theta(\overline{a})\\
C_\tau[\![\mathsf{App}\ E]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathsf{App}\ (C_\tau[\![E]\!]^\Gamma_\theta(\varepsilon))(\overline{a})\\
C_\tau[\![E\ \phi]\!]^\Gamma_\theta(\overline{a}) &\triangleq C_\tau[\![E]\!]^\Gamma_\theta(\overline{a})\\
C_\tau[\![\Lambda\chi.E]\!]^\Gamma_\theta(\overline{a}) &\triangleq C_\tau[\![E]\!]^{\Gamma,\chi}_\theta(\overline{a})
\end{aligned}
$$

The most complicated cases are for **let**-expressions, which rely on the types of the variable binding to determine the difference between strictness and laziness. These cases are defined (top-to-bottom) as follows:

$$
\begin{aligned}
C_\tau[\![\mathbf{let}\,x{:}\sigma = R\,\mathbf{in}\,E]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathbf{let}\,x^{\mathsf{U}}_{\mathsf{PtrR}} = \mathcal{A}[\![R]\!]^\Gamma_\theta\ \mathbf{in}\,C_\tau[\![E]\!]^{\Gamma,x:\sigma}_{[x_{\mathsf{PtrR}}/x]\theta}(\overline{a})\\
C_\tau[\![\mathbf{let}\,x{:}\sigma = e\,\mathbf{in}\,E]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathbf{let}\,x^{lev(\eta)}_{\mathsf{PtrR}} = \mathcal{E}_\eta[\![e]\!]^\Gamma_\theta\ \mathbf{in}\,C_\tau[\![E]\!]^{\Gamma,x:\sigma}_{[x_{\mathsf{PtrR}}/x]\theta}(\overline{a})\\
&\qquad (\text{if } \Gamma\vdash e\ \mathbf{bind}\text{ and }\Gamma\vdash\sigma \overset{conv}{\rightsquigarrow} \eta)\\
C_\tau[\![\mathbf{let}\,x{:}\sigma = E\,\mathbf{in}\,e]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathbf{let}\,x^{\mathsf{U}}_{\pi} = C_\tau[\![E]\!]^\Gamma_\theta(\varepsilon)\ \mathbf{in}\,C[\![e]\!]^{\Gamma,x:\sigma}_{[x_\pi/x]\theta}(\overline{a})\\
&\qquad (\text{if } \Gamma\vdash\sigma \overset{conv}{\rightsquigarrow} \mathtt{Eval}^{\mathsf{U}}\text{ and }\Gamma\vdash\sigma \overset{rep}{\rightsquigarrow} \pi)\\
C_\tau[\![\mathbf{let}\,x{:}\sigma = E_2\,\mathbf{in}\,E_1[x]]\!]^\Gamma_\theta(\overline{a}) &\triangleq \mathbf{let}\,x^{\mathsf{L}}_{\mathsf{PtrR}} = C_\tau[\![E]\!]^\Gamma_\theta(\varepsilon)\ \mathbf{in}\,C_\sigma[\![E_1]\!]^{\Gamma,x:\sigma}_{[x_{\mathsf{PtrR}}/x]\theta}(\overline{a})[x_{\mathsf{PtrR}}]\\
&\qquad (\text{if } \Gamma\vdash\sigma \overset{conv}{\rightsquigarrow} \mathtt{Eval}^{\mathsf{L}})
\end{aligned}
$$

Note that $\mathcal{IL}$ closing, frame, and binding contexts are all special cases of evaluation contexts, and are defined as above, and all translate to their corresponding special case in $\mathcal{ML}$.

Lemma 29 (Context Compilation). *If* $\Gamma\vdash E\ @\ e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ *then*

$$C[\![E[e]]\!]^\Gamma_\theta(\overline{a}) \triangleq C_\tau[\![E]\!]^\Gamma_\theta(\overline{a})[C[\![e]\!]^\Gamma_\theta(\varepsilon)]$$

Proof. By induction on the derivation of $\Gamma\vdash E\ @\ e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$. □

Lemma 30 (Instantiation). *(1) For any* $\Gamma, x:\tau\vdash e:\sigma$ *and* $\Gamma\vdash a':\tau$, *renaming an argument for a variable commutes with compilation:* $C[\![e[a'/x]]\!]^\Gamma_\theta(\overline{a}) \triangleq C[\![e]\!]^{\Gamma,x:\tau}_{[\mathcal{A}[\![a']\!]^\Gamma_\theta/x]\theta}(\overline{a})$.

(2) For any $\Gamma,\chi\vdash e:\sigma$ *and* $\Gamma\vdash[\phi/\chi]$ **poly**, *instantiation of a type variable commutes with compilation:* $C[\![e[\phi/\chi]]\!]^\Gamma_\theta(\overline{a}) \triangleq C[\![e]\!]^{\Gamma,\chi}_\theta(\overline{a})$.

Proof. By induction on the syntax of e. □

Lemma 31 (Forward Simulation). *For any $\mathcal{IL}$ expression e_i and $\mathcal{ML}$ expression e_m,*

(1) If $\Gamma\vdash_C e_i \sim e_m : \tau$ *and* $\Gamma\vdash e_i \mapsto e_i' : \tau$ *by a* $\beta_{\rightsquigarrow}$, $\beta_\forall$, *rename, name, or* $name_{\mathtt{I\#}}$ *step, then* $\Gamma\vdash e_i' \sim e_m : \tau$.

(2) If $\Gamma\vdash_C e_i\sim e_m : \tau$ *and* $e_i\mapsto e_i'$ *by a* $\beta_{\{\}}$, $\beta_{\mathtt{Int}}$, *look, or comm step, then* $e_m\mapsto e_m'$ *and* $\Gamma\vdash e_i'\sim e_m:\tau$ *for some* e_m'.

Therefore, if $\Gamma\vdash_C e_i\sim e_m:\tau$ *and* $\Gamma\vdash e_i\mapsto^* e_i':\tau$ *then* $e_m\mapsto^* e_m'$ *and* $\Gamma\vdash e_i'\sim e_m':\tau$ *for some* e_m'.

Proof. First consider the cases where $\Gamma\vdash e_i\mapsto e_i':\tau$ by applying a single step of the reduction rules directly (*i.e.,* not *compat*). The first case of $\mathcal{IL}$ reductions, which are erased by compilation are:

- $\beta_{\rightsquigarrow}$ $\mathcal{C}[\![(\lambda x{:}\tau.e)\ a']\!]^{\Gamma}_{\theta}(\overline{a}) \triangleq \mathcal{C}[\![\lambda x{:}\tau.e]\!]^{\Gamma}_{\theta}(\mathcal{A}[\![a']\!]^{\Gamma}_{\theta}, \overline{a}) \triangleq \mathcal{C}[\![e]\!]^{\Gamma,x:\tau}_{[\mathcal{A}[\![a']\!]^{\Gamma}_{\theta}/x]\theta}(\overline{a}) \triangleq \mathcal{C}[\![e[a'/x]]\!]^{\Gamma}_{\theta}(\overline{a})$ which follows from Lemma 30.
- *rename* is the same as $\beta_{\rightsquigarrow}$, via the encoding of **let** as applied λ-abstractions.
- $\beta_{\forall}$ is $\mathcal{C}[\![(\Lambda\chi.e)\ \phi]\!]^{\Gamma}_{\theta}(\overline{a}) \triangleq \mathcal{C}[\![\Lambda\chi.e]\!]^{\Gamma}_{\theta}(\overline{a}) \triangleq \mathcal{C}[\![e]\!]^{\Gamma,\chi}_{\theta}(\overline{a}) \triangleq \mathcal{C}[\![e]\!]^{\Gamma}_{\theta}(\overline{a})$ which follows from Lemma 30.
- $name_{\mathtt{I\#}}$ is $\mathcal{C}[\![\mathtt{I\#}^{\gamma}\ e]\!]^{\Gamma}_{\theta}(\varepsilon) \triangleq \mathbf{let}\ x^{\mathsf{U}}_{\mathtt{IntR}} = \mathcal{C}[\![e]\!]^{\Gamma}_{\theta}(\varepsilon)\ \mathbf{in}\ \mathtt{I\#}(x_{\mathtt{IntR}}) \triangleq \mathcal{C}[\![\mathbf{let}\ x{:}\mathtt{Int\#} = e\ \mathbf{in}\ \mathtt{I\#}^{\gamma}\ x]\!]^{\Gamma}_{\theta}(\varepsilon)$.
- *name* is similar to $name_{\mathtt{I\#}}$, and depends on the argument of application and its kind.

The second case of $\mathcal{IL}$ reductions that are mirrored by compilation are:

- $\beta_{\{\}}$ is

$$\begin{aligned}\mathcal{C}[\![\mathtt{App}\ (\mathtt{Clos}^{\gamma}\ e)]\!]^{\Gamma}_{\theta}(\overline{a}) &\triangleq \mathtt{App}\ (\mathtt{Clos}^{n}\ \mathcal{E}_{\eta}[\![e]\!]^{\Gamma}_{\theta})(\overline{a})\\ &\mapsto_{Apply} \mathcal{E}_{\eta}[\![e]\!]^{\Gamma}_{\theta}(\overline{a})\\ &\mapsto^{?}_{Call} \mathcal{C}[\![e]\!]^{\Gamma}_{\theta}(\overline{a})\end{aligned}$$

  assuming $\Gamma \vdash e : \tau$ and $\Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta$, which forces $\overline{\pi} = arity(\eta)$ and $n = |\overline{\pi}| = |\overline{a}|$ and $\overline{a} \overset{rep}{\rightsquigarrow} \overline{\pi}$.
- $\beta_{\mathtt{Int}}$ follows from Lemma 30

$$\begin{aligned}\mathcal{C}[\![\mathbf{case}\ \mathtt{I\#}^{\gamma}\ a'\ \mathbf{of}\ \mathtt{I\#}\ x \to e]\!]^{\Gamma}_{\theta}(\overline{a}) &\triangleq \mathbf{case}\ \mathtt{I\#}(\mathcal{A}[\![a']\!]^{\Gamma}_{\theta})\ \mathbf{of}\ \mathtt{I\#}(x_{\mathtt{IntR}}) \to \mathcal{C}[\![e]\!]^{\Gamma,x:\mathtt{Int\#}}_{[x_{\mathtt{IntR}}/x]\theta}(\overline{a})\\ &\mapsto_{Unbox} \mathcal{C}[\![e]\!]^{\Gamma,x:\mathtt{Int\#}}_{[\mathcal{A}[\![a']\!]^{\Gamma}_{\theta}/x]\theta}(\overline{a}) \triangleq \mathcal{C}[\![e[a'/x]]\!]^{\Gamma}_{\theta}(\overline{a})\end{aligned}$$

- *comm* is, for some ν and ψ

$$\begin{aligned}\mathcal{C}[\![F[\mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ Fin]]\!]^{\Gamma}_{\theta}(\overline{a}) &\hookleftarrow^{*}_{Unlift} \mathcal{C}_{\sigma}[\![F]\!]^{\Gamma}_{\theta}(\overline{a})[\mathbf{let}\ x^{\psi}_{\mathtt{PtrR}} = \mathcal{E}_{\nu}[\![e]\!]^{\Gamma}_{\theta}\ \mathbf{in}\ \mathcal{C}[\![Fin]\!]^{\Gamma}_{\theta}(\varepsilon)]\\ &\mapsto_{Alloc} \mathbf{let}\ x^{\psi}_{\mathtt{PtrR}} = \mathcal{E}_{\nu}[\![e]\!]^{\Gamma}_{\theta}\ \mathbf{in}\ \mathcal{C}_{\sigma}[\![F]\!]^{\Gamma}_{\theta}(\overline{a})[\mathcal{C}[\![Fin]\!]^{\Gamma}_{\theta}(\varepsilon)]]\\ &\triangleq \mathcal{C}[\![\mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ F[Fin]]\!]^{\Gamma}_{\theta}(\overline{a})\end{aligned}$$

  where $Alloc = LAlloc$ if $\psi = \mathsf{L}$ and $Alloc = SAlloc$ if $\psi = \mathsf{U}$.
- *look* is, by Lemma 27, for some ν and ψ

$$\begin{aligned}\mathcal{C}[\![\mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ E[x]]\!]^{\Gamma}_{\theta}(\overline{a}) &\hookleftarrow^{*}_{Unlift} \mathbf{let}\ x^{\psi}_{\mathtt{PtrR}} = \mathcal{E}_{\nu}[\![e]\!]^{\Gamma}_{\theta}\ \mathbf{in}\ \mathcal{C}_{\sigma}[\![E]\!]^{\Gamma}_{\theta}(\overline{a})[x_{\mathtt{PtrR}}]\\ &\mapsto_{Inline} \mathbf{let}\ x^{\mathsf{U}}_{\mathtt{PtrR}} = \mathcal{E}_{\nu}[\![e]\!]^{\Gamma}_{\theta}\ \mathbf{in}\ \mathcal{C}_{\sigma}[\![E]\!]^{\Gamma}_{\theta}(\overline{a})[\mathcal{E}_{\nu}[\![e]\!]^{\Gamma}_{\theta}]\\ &\triangleq \mathcal{C}[\![\mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ E[e]]\!]^{\Gamma}_{\theta}(\overline{a})\end{aligned}$$

  where $Inline = Look$ or $Inline = Memo$ if $\nu = \mathtt{Eval}^{\gamma}$ and $Inline = Fun$ if $\nu = \mathtt{Call}[\overline{\pi}]$.

Now, assume that $\Gamma \vdash e \mapsto e' : \tau$ by applying one of the reduction rules as in the above steps. Reduction within an evaluation context $E[e] \mapsto E[e']$ proceeds like so:

$$\begin{aligned}\mathcal{C}[\![E[e]]\!]^{\Gamma}_{\theta}(\overline{a}) &\triangleq \mathcal{C}_{\tau}[\![E]\!]^{\Gamma}_{\theta}(\overline{a})[\mathcal{C}[\![e]\!]^{\Gamma}_{\theta}(\varepsilon)] \hookleftarrow^{*}_{Unlift} E''[e'']\\ &\mapsto \mathcal{C}_{\tau}[\![E]\!]^{\Gamma}_{\theta}(\overline{a})[\mathcal{C}[\![e']\!]^{\Gamma}_{\theta}(\varepsilon)]\\ &\mapsto^{*}_{Unlift} \mathcal{C}[\![E[e']]\!]^{\Gamma}_{\theta}(\overline{a})\end{aligned}$$

where additional applications of the *Unlift* step may be needed when a bound lazy expression (of kind $\mathtt{Eval}^{\mathsf{L}}$) is finally reduced to a reference R. □

Lemma 32 (Backward Simulation). *If* $\Gamma \vdash_{C} e_i \sim e_m : \tau$ *and* $e_m \mapsto^{*} e'_m$ *then* $\Gamma \vdash e_i \mapsto^{*} e'_i : \tau$ *and* $\Gamma \vdash_{C} e'_i \sim e'_m$ *for some* e'.

Proof. Note that if $\Gamma \vdash_{C} e_i \sim e_m : \tau$ then $e_m \not\mapsto$ by the *Call* or *Move* steps. Otherwise, given $e_m \mapsto e'_m$ by any other step, it is the case that $\Gamma \vdash e_i \mapsto^{*} e''_i : \tau$ by $\beta_{\rightsquigarrow}$, $\beta_{\forall}$, *rename*, *name*, and $name_{\mathtt{I\#}}$ steps, followed by $\Gamma \vdash e''_i \mapsto e'_i : \tau$ by one of the other steps. Therefore, by forward simulation Lemma 25 and determinism of the $\mathcal{ML}$ operational semantics, we know that $\Gamma \vdash_{C} e'_i \sim e'_m$. □

Types σ corresponding to the source call-by-name System F

$$\sigma ::= a \mid \mathtt{Int}^{\gamma} \mid {}^{\gamma}\{\tau\} \mid \forall t{:}\kappa.\sigma \qquad \tau ::= \sigma \rightsquigarrow \sigma' \qquad \kappa ::= \mathsf{TYPE\ PtrR\ Eval}^{\gamma} \qquad \gamma ::= \mathsf{L}$$

Types σ corresponding to the source call-by-value System F

$$\tau ::= a \mid \mathtt{Int}^{\gamma} \mid {}^{\gamma}\{\tau\} \qquad \tau ::= \sigma \rightsquigarrow \sigma' \mid \forall t{:}\kappa.\sigma \qquad \kappa ::= \mathsf{TYPE\ PtrR\ Eval}^{\gamma} \qquad \gamma ::= \mathsf{U}$$

Decompilation of source types $\boxed{[\![\sigma]\!]^{-1}}$ and target-only types $\boxed{[\![\tau]\!]^{-1}}$

$$[\![a]\!]^{-1} \triangleq a \qquad [\![\mathtt{Int}^{\gamma}]\!]^{-1} \triangleq \mathtt{Int} \qquad [\![{}^{\gamma}\{\tau\}]\!]^{-1} \triangleq [\![\tau]\!]^{-1}$$

$$[\![\forall t{:}\kappa.\sigma]\!]^{-1} \triangleq \forall t.[\![\sigma]\!]^{-1} \qquad [\![\sigma \rightsquigarrow \sigma']\!]^{-1} \triangleq [\![\sigma]\!]^{-1} \to [\![\sigma']\!]^{-1}$$

Fig. 9. Decompiling $\mathcal{IL}$ types back to System F.

B.5 Correctness of closed compilation

The correspondence between the equational theory of $\mathcal{IL}$ and the abstract machine of $\mathcal{ML}$ is based on four parts:

(1) Standardization and confluence relating the equational theory of $\mathcal{IL}$ to the call-by-name operational semantics of $\mathcal{IL}$ (defined as a sub-relation of the equational theory).
(2) A bisimulation between the call-by-name and call-by-need operational semantics of $\mathcal{IL}$ based on *unwinding* (*i.e.,* copying) **let**-bindings.
(3) A bisimulation between the call-by-need operational semantics of $\mathcal{IL}$ and the operational semantics of $\mathcal{ML}$ based on the compilation given in Fig. 8.
(4) A bisimulation between the operational semantics and abstract machine of $\mathcal{ML}$.

Each of these four parts are bi-directional, and bisimulations compose with one another. Therefore, we can trace high-level equalities in $\mathcal{IL}$ all the way down to the $\mathcal{ML}$ machine, and back. The only restriction imposed is the types of final answers allowed: a necessary restriction to respect the full η-extensionality we're after.

Theorem 8 (Soundness and Completeness)**.**

(1) For any $\vdash e : \mathtt{Int\#}$, $\vdash e = i : \mathtt{Int\#}$ *if and only if* $\langle \mathcal{E}_{\mathtt{Eval}^{\mathsf{U}}}[\![e]\!] \mid \varepsilon \mid \varepsilon \rangle \mapsto^{*} \langle i \mid \varepsilon \mid H \rangle$.
(2) For any $\vdash e : \mathtt{Int}^{\gamma}$, $\vdash e = \mathtt{I\#}^{\gamma}\, i : \mathtt{Int}^{\gamma}$ *if and only if* $\langle \mathcal{E}_{\mathtt{Eval}^{\gamma}}[\![e]\!] \mid \varepsilon \mid \varepsilon \rangle \mapsto^{*} \langle \mathtt{I\#}(i) \mid \varepsilon \mid H \rangle$.

Proof. We show the case for $\mathtt{Int\#}$ as $\mathtt{Int}^{\gamma}$ is analogous.
First, from the assumption that $\vdash e = i : \mathtt{Int\#}$, we know:

- $\vdash e \mapsto^{*} i : \mathtt{Int\#}$ in call-by-name $\mathcal{IL}$ from Corollary 4,
- $\vdash e \mapsto^{*} H[i] : \mathtt{Int\#}$ in call-by-need $\mathcal{IL}$ from Lemmas 12 and 15 and the fact that i is closed,
- $\mathcal{E}_{\mathtt{Eval}^{\mathsf{U}}}[\![e]\!] \mapsto^{*} H'[i]$ in $\mathcal{ML}$ from Lemmas 28 and 31, and
- $\langle \mathcal{E}_{\mathtt{Eval}^{\mathsf{U}}}[\![e]\!] \mid \varepsilon \mid \varepsilon \rangle \mapsto^{*} \langle i \mid \varepsilon \mid H \rangle$ from Lemma 25 and Corollary 5.

Second, from the assumption that $\langle \mathcal{E}_{\mathtt{Eval}^{\mathsf{U}}}[\![e]\!] \mid \varepsilon \mid \varepsilon \rangle \mapsto^{*} \langle i \mid \varepsilon \mid H \rangle$, we know:

- $\mathcal{E}_{\mathtt{Eval}^{\mathsf{U}}}[\![e]\!] \mapsto^{*} H[i]$ in $\mathcal{ML}$ from Lemma 26 and Corollary 5,
- $\vdash e \mapsto^{*} H'[i] : \mathtt{Int\#}$ in call-by-need $\mathcal{IL}$ from Lemmas 28 and 32,
- $\vdash e \mapsto^{*} i : \mathtt{Int\#}$ in call-by-name $\mathcal{IL}$ from Lemmas 12 and 13 and the fact that i is closed, and
- $\vdash e = i : \mathtt{Int\#}$ from Theorem 5.

□

Expressions corresponding to call-by-name -value System F (invariant: $x : \sigma$)

$$e ::= x \mid \mathtt{I\#}^{\gamma}\, i \mid e\ e' \mid \lambda x{:}\sigma.e \mid e\ \sigma \mid \Lambda t{:}\kappa.e \mid \mathsf{App}\ e \mid \mathtt{Clos}^{\gamma}\, S$$

Decompilation of serious expressions $\boxed{[\![e]\!]^{-1}}$ (invariant: $e : \tau$ or $e : \sigma$)

$$[\![x]\!]^{-1} \triangleq x \qquad [\![\mathtt{I\#}^{\gamma}\, i]\!]^{-1} \triangleq i$$
$$[\![e\ e']\!]^{-1} \triangleq [\![e]\!]^{-1}\ [\![e']\!]^{-1} \qquad [\![\lambda x{:}\sigma.e]\!]^{-1} \triangleq \lambda x{:}\sigma.[\![e]\!]^{-1}$$
$$[\![e\ \sigma]\!]^{-1} \triangleq [\![e]\!]^{-1}\ [\![\sigma]\!]^{-1} \qquad [\![\Lambda t{:}\kappa.e]\!]^{-1} \triangleq \Lambda t.[\![e]\!]^{-1}$$
$$[\![\mathsf{App}\ e]\!]^{-1} \triangleq [\![e]\!]^{-1} \qquad [\![\mathtt{Clos}^{\gamma}\, V]\!]^{-1} \triangleq \eta[\![V]\!]^{-1}$$

η-expanded decompilation of values $\boxed{\eta[\![e]\!]^{-1}}$ (invariant: $e : \tau$ and e is a value)

$$\eta[\![\lambda x{:}\sigma.e]\!]^{-1} \triangleq \lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1} \qquad \eta[\![\mathsf{App}\ e]\!]^{-1} \triangleq \lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}\ x \qquad (\text{if } \mathsf{App}\ e : \sigma \rightsquigarrow \sigma')$$
$$\eta[\![\Lambda t{:}\kappa.e]\!]^{-1} \triangleq \Lambda t.[\![e]\!]^{-1} \qquad \eta[\![\mathsf{App}\ e]\!]^{-1} \triangleq \Lambda t.[\![e]\!]^{-1}\ t \qquad (\text{if } \mathsf{App}\ e : \forall t{:}\kappa.\sigma)$$

Fig. 10. Decompiling $\mathcal{IL}$ expressions back to System F.

C CORRECTNESS OF SYSTEM F-TO-$\mathcal{IL}$ COMPILATION

Decompilation of types and expressions from $\mathcal{IL}$ back to System F is shown in Figs. 9 and 10 for call-by-name and call-by-value, respectively, which is defined over the sublanguage of $\mathcal{IL}$ that is reachable from compiling System F. Note that this sublanguage is closed under reduction.

We now show that both call-by-name and call-by-value compilation a decompilation form an equational correspondence between System F and $\mathcal{IL}$.

Lemma 33 (η-Expansion). *For both the call-by-value and call-by-name $\mathcal{IL}$ sublanguages:*

(1) if $\Gamma \vdash e : \sigma \rightsquigarrow \sigma'$, then $\eta[\![e]\!]^{-1} =_{\beta_{\{\}}} \lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}\ x$, and
(2) if $\Gamma \vdash e : \forall t{:}\kappa.\sigma$, then $\eta[\![e]\!]^{-1} =_{\beta_{\forall}} \Lambda t{:}\kappa.[\![e]\!]^{-1}\ t$.

Proof. By cases on the form of e. First, note that either type of $\eta[\![\mathsf{App}\ e]\!]^{-1}$ is definitionally equal to the right-hand side. If instead e is a λ- or Λ-abstraction, then $\eta[\![e]\!]^{-1}$ β-expands to the right-hand side. □

Lemma 34 (Value Preservation). *For both call-by-name and call-by-value:*

(1) Given any value $\Gamma \vdash V : \tau$ in System F, $[\![\Gamma]\!] \vdash [\![V]\!]$ **subst** *in $\mathcal{IL}$.*
(2) Given any $\Gamma \vdash e$ **subst** *in the $\mathcal{IL}$ sublanguage, $[\![e]\!]^{-1}$ is a value in System F.*
(3) Given any $\Gamma \vdash e$ **subst** *in the $\mathcal{IL}$ sublanguage, $\eta[\![e]\!]^{-1}$ is a value in System F.*

Proof. Call-by-name value preservation is immediate, because every expression is a value in call-by-name System F, and every System F expression compiles to one with a type of kind $\mathsf{TYPE\ PtrR\ Eval}^{\mathsf{L}}$ so it is a value in $\mathcal{IL}$.

Call-by-value value preservation follows by cases on the forms of values in the source System F and target $\mathcal{IL}$ sublanguage. Note that the η-expansion of $\eta[\![\mathsf{App}\ e]\!]^{-1}$ ensures that this is always a value for types like $\sigma \rightsquigarrow \sigma'$, from which value preservation follows for $[\![e]\!]^{-1}$ for expressions of type σ which all have kind $\mathsf{TYPE\ PtrR\ Eval}^{\mathsf{U}}$. □

Lemma 35 (Forward Inverse). *In call-by-value or call-by-name System F:*

(1) $[\![[\![\tau]\!]]\!]^{-1} \triangleq \tau$, and
(2) if $\Gamma \vdash e : \tau$ then $[\![[\![e]\!]]\!]^{-1} \triangleq e$.

Proof. By induction on the syntax of τ and e, each case following directly from the inductive hypothesis. $\square$

Lemma 36 (Backward Inverse). *In the $\mathcal{IL}$ sublanguage, for both call-by-value and call-by-name (de)compilation:*

(1) $[\![[\![\tau]\!]^{-1}]\!] \triangleq \tau$ *and* $[\![[\![\sigma]\!]^{-1}]\!] \triangleq {}^{Y}\{\sigma\}$,
(2) if $\Gamma \vdash e : \sigma$ *then* $[\![[\![e]\!]^{-1}]\!] =_{\beta_{\{\}}\eta_{\rightsquigarrow}\eta_{\forall}} e$,
(3) if $\Gamma \vdash e : \tau$ *then* $\mathsf{App}\ [\![[\![e]\!]^{-1}]\!] =_{\beta_{\{\}}\eta_{\rightsquigarrow}\eta_{\forall}} e$, *and*
(4) if $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e$ **subst** *then* $[\![\eta[\![e]\!]^{-1}]\!] =_{\beta_{\{\}}\eta_{\rightsquigarrow}\eta_{\forall}} \mathtt{Clos}^{Y}\ e$.

Proof. The first part follows directly on the syntax of τ and σ, and the remaining parts follow simultaneously by induction on the structure of the syntax of e. The cases for x and $\mathtt{I\#}^{Y}\ i$ are immediate, and the remaining cases are as follows:

- (2) $[\![[\![e\ e']\!]^{-1}]\!] \triangleq \mathsf{App}\ [\![[\![e]\!]^{-1}]\!]\ [\![[\![e']\!]^{-1}]\!] =_{(3)} e\ [\![[\![e']\!]^{-1}]\!] =_{(2)} e\ e'$
- (2) $[\![[\![e\ \sigma]\!]^{-1}]\!] =_{(2,3)} e\ \sigma$ is analogous to the previous case for call-by-value, and for call-by-name is: $[\![[\![e\ \sigma]\!]^{-1}]\!] \triangleq [\![[\![e]\!]^{-1}]\!]\ [\![[\![\sigma]\!]^{-1}]\!] =_{(2)} e\ [\![[\![\sigma]\!]^{-1}]\!] =_{(1)} e\ \sigma$
- (2) $[\![[\![\Lambda t{:}\kappa.e]\!]^{-1}]\!] \triangleq \Lambda t{:}\kappa.[\![[\![e]\!]^{-1}]\!] =_{(2)} \Lambda t{:}\kappa.e$ Note this case only applies in call-by-name (de)compilation.
- (2) $[\![[\![\mathtt{Clos}^{Y}\ S]\!]^{-1}]\!] \triangleq [\![\eta[\![S]\!]^{-1}]\!] =_{(4)} \mathtt{Clos}^{Y}\ S$
- (3) $\mathsf{App}\ [\![[\![\lambda x{:}\sigma.e]\!]^{-1}]\!] \triangleq \mathsf{App}\ [\![\lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}]\!] \triangleq \mathsf{App}\ (\mathtt{Clos}^{Y}\ \lambda x{:}[\![[\![\sigma]\!]^{-1}]\!].[\![[\![e]\!]^{-1}]\!])$
  $=_{(1,2)} \mathsf{App}\ (\mathtt{Clos}^{Y}\ \lambda x{:}\sigma.e) =_{\beta_{\{\}}} \lambda x{:}\sigma.e$
- (3) $\mathsf{App}\ [\![[\![\Lambda t{:}\kappa.e]\!]^{-1}]\!] =_{(1,2)\beta_{\{\}}} \Lambda t{:}\kappa.e$ follows analogously to the previous case. Note this case only applies in call-by-value (de)compilation.
- (3) $\mathsf{App}\ [\![[\![\mathsf{App}\ e]\!]^{-1}]\!] \triangleq \mathsf{App}\ [\![[\![e]\!]^{-1}]\!] =_{(2)} \mathsf{App}\ e$
- (4) $[\![\eta[\![\lambda x{:}\sigma.e]\!]^{-1}]\!] \triangleq [\![\lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}]\!] \triangleq \mathtt{Clos}^{Y}\ \lambda x{:}[\![[\![\sigma]\!]^{-1}]\!].[\![[\![e]\!]^{-1}]\!] =_{(1,2)} \mathtt{Clos}^{Y}\ \lambda x{:}\sigma.e$
- (4) $[\![\eta[\![\Lambda t{:}\kappa.e]\!]^{-1}]\!] =_{(1,2)} \mathtt{Clos}^{Y}\ \Lambda t{:}\kappa.e$ follows analogously to the previous case. Note this case only applies in call-by-value (de)compilation.
- (4) Given $\Gamma \vdash e : {}^{Y}\{\sigma \rightsquigarrow \sigma'\}$:
$$[\![\eta[\![\mathsf{App}\ e]\!]^{-1}]\!] \triangleq [\![\lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}\ x]\!] \triangleq \mathtt{Clos}^{Y}\ (\lambda x{:}[\![[\![\sigma]\!]^{-1}]\!].\mathsf{App}\ [\![[\![e]\!]^{-1}]\!]\ x)$$
$$=_{(1,2)} \mathtt{Clos}^{Y}\ (\lambda x{:}\sigma.\mathsf{App}\ e\ x) =_{\eta_{\rightsquigarrow}} \mathtt{Clos}^{Y}\ (\mathsf{App}\ e)$$
- Given $\Gamma \vdash e : {}^{Y}\{\forall t{:}\kappa.\sigma\}$, $[\![\eta[\![\mathsf{App}\ e]\!]^{-1}]\!] =_{(1,2)\eta_{\forall}} \mathsf{App}\ e$ analogously to the previous case. $\square$

Lemma 37 (Forward Soundness). *For any* $\Gamma \vdash e_i : \tau$ *in System F, if* $\Gamma \vdash e_1 = e_2 : \tau$ *in either call-by-value or call-by-name then* $[\![\Gamma]\!] \vdash [\![e_1]\!] = [\![e_2]\!] : [\![\tau]\!]$ *with regards to the corresponding compilation.*

Proof. Note that, since the compilation translation is compositional, substitution commutes with compilation (*i.e.,* $[\![e]\!][[\![V]\!]/x] \triangleq [\![e[V/x]]\!]$ and similarly for type substitution). The equational axioms of call-by-name System F are sound w.r.t. compilation as follows:

- $(\beta_{\{\}})$
$$[\![(\lambda x{:}\tau.e)\ V]\!] \triangleq \mathsf{App}\ (\mathtt{Clos}^{Y}\ \lambda x{:}[\![\tau]\!].[\![e]\!])\ [\![V]\!]$$
$$=_{\beta_{\{\}}} (\lambda x{:}[\![\tau]\!].[\![e]\!])\ [\![V]\!] =_{\beta_{\rightsquigarrow}} [\![e]\!]\ [[\![V]\!]/x] \triangleq [\![e[V/x]]\!]$$
- $(\beta_{\forall})$
$$[\![(\Lambda t.e)\ \tau]\!] \triangleq (\Lambda t{:}\kappa.[\![e]\!])\ [\![\tau]\!] =_{\beta_{\forall}} [\![e]\!][[\![\tau]\!]/t] \triangleq [\![e[\tau/t]]\!]$$

- $(\eta_{\{\}})$

$$[\![\lambda x{:}\tau.(V\ x)]\!] \triangleq \texttt{Clos}^{\gamma}\ \lambda x{:}[\![\tau]\!].(\texttt{App}\ [\![V]\!]\ x) =_{\eta_{\rightsquigarrow}} \texttt{Clos}^{\gamma}\ (\texttt{App}\ [\![V]\!]) =_{\eta_{\{\}}} [\![V]\!]$$

- $(\eta_{\forall})$

$$[\![\Lambda t.V\ t]\!] \triangleq \Lambda t.[\![V]\!]\ t =_{\eta_{\forall}} [\![V]\!]$$

- $(name)$

$$\begin{aligned}[\![(\lambda x{:}\tau.e\ x)\ e']\!] &\triangleq \texttt{App}\ (\texttt{Clos}^{\gamma}\ \lambda x{:}[\![\tau]\!].\texttt{App}\ [\![e]\!]\ x)\ [\![e']\!] \\ &=_{\beta_{\{\}}} (\lambda x{:}[\![\tau]\!].\texttt{App}\ [\![e]\!]\ x)\ [\![e']\!] =_{\eta_{\rightsquigarrow}} \texttt{App}\ [\![e]\!]\ [\![e']\!] \triangleq [\![e\ e']\!]\end{aligned}$$

The equational axioms of call-by-value System F are similarly sound, where the only difference is that the compilation of the $\beta_{\forall}$ and $\eta_{\forall}$ more closely resemble $\beta_{\{\}}$ and $\eta_{\{\}}$. □

Lemma 38 (Backward Soundness)**.** *In either the call-by-value or call-by-name sublanguage of $\mathcal{IL}$:*

(1) for any $\Gamma \vdash e_i : \sigma$, if $\Gamma \vdash e_1 = e_2 : \sigma$ then $[\![\Gamma]\!]^{-1} \vdash [\![e_1]\!]^{-1} = [\![e_2]\!]^{-1} : [\![\sigma]\!]^{-1}$, and
(2) for any $\Gamma \vdash e_i : \tau$, if $\Gamma \vdash e_1 = e_2 : \tau$ then $[\![\Gamma]\!]^{-1} \vdash \eta[\![e_1]\!]^{-1} = \eta[\![e_2]\!]^{-1} : [\![\tau]\!]^{-1}$,

in the corresponding System F.

Proof. Note that, since the compilation translation is compositional, substitution commutes with decompilation (*i.e.*, $[\![e]\!]^{-1}[[\![e']\!]^{-1}/x] \triangleq [\![e[e'/x]]\!]^{-1}$ and similarly for type substitution). The equational axioms of the call-by-name $\mathcal{IL}$ sublanguage sound w.r.t. decompilation as follows:

- $(\beta_{\rightsquigarrow})$

$$[\![(\lambda x{:}\tau.e)\ e']\!]^{-1} \triangleq (\lambda x{:}\tau.[\![e]\!]^{-1})\ [\![e']\!]^{-1} =_{\beta_{\{\}}} [\![e]\!]^{-1}[[\![e']\!]^{-1}/x] = [\![e[e'/x]]\!]^{-1}$$

- $(\beta_{\forall})$

$$[\![(\Lambda t{:}\kappa.e)\ \tau]\!]^{-1} \triangleq (\Lambda t{:}\kappa.[\![e]\!]^{-1})\ [\![\tau]\!]^{-1} =_{\beta_{\forall}} [\![e]\!]^{-1}[[\![\tau]\!]^{-1}/t] = [\![e[\tau/t]]\!]^{-1}$$

- $(\beta_{\{\}})$ Given that $\Gamma \vdash e : \sigma \rightsquigarrow \sigma'$, we have

$$\eta[\![\texttt{App}\ (\texttt{Clos}^{\gamma}\ e)]\!]^{-1} \triangleq \lambda x{:}[\![\sigma]\!]^{-1}.[\![\texttt{Clos}^{\gamma}\ e]\!]^{-1}\ x \triangleq \lambda x{:}[\![\sigma]\!]^{-1}.\eta[\![e]\!]^{-1}\ x =_{\eta_{\{\}}} \eta[\![e]\!]^{-1}$$

  The case in call-by-value where $\Gamma \vdash e : \forall t{:}\kappa.\sigma$ is analogous.
- $(\eta_{\rightsquigarrow})$

$$\eta[\![\lambda x{:}\sigma.(e\ x)]\!]^{-1} \triangleq \lambda x{:}[\![\sigma]\!]^{-1}.([\![e]\!]^{-1}\ x) =_{\beta_{\{\}}} \eta[\![e]\!]^{-1}$$

- $(\eta_{\forall})$ In call-by-name, we have

$$[\![\Lambda t{:}\kappa.(e\ t)]\!]^{-1} \triangleq \Lambda t.([\![e]\!]^{-1}\ t) =_{\eta_{\forall}} [\![e]\!]^{-1}$$

  whereas in call-by-value, we have an analogous equality to the previous case.
- $(\eta_{\{\}})$ Given that $\Gamma \vdash V : {}^{\gamma}\{\sigma \rightsquigarrow \sigma'\}$

$$[\![\texttt{Clos}^{\gamma}\ (\texttt{App}\ e)]\!]^{-1} \triangleq \eta[\![\texttt{App}\ e]\!]^{-1} \triangleq \lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}\ x =_{\eta_{\{\}}} [\![V]\!]^{-1}$$

  The case in call-by-value where $\Gamma \vdash e : {}^{\gamma}\{\forall t{:}\kappa.\sigma\}$ is analogous.

Congruence of equality follows by induction on the syntax of expressions, where the only cases that do not follow immediately from the inductive hypothesis are those which introduce $[\![e]\!]$ when $\Gamma \vdash e : \tau$. This can occur with function application and $\texttt{App}\ e$ expansion of type $\sigma \rightsquigarrow \sigma'$, which simplify to a known case as follows:

$$[\![e\ e']\!]^{-1} \triangleq [\![e]\!]^{-1}\ [\![e']\!]^{-1} =_{name} (\lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}\ x)\ [\![e']\!]^{-1} = \eta[\![e]\!]^{-1}\ [\![e']\!]^{-1}$$

$$\eta[\![\texttt{App}\ e]\!]^{-1} \triangleq \lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}\ x =_{\beta_{\{\}}} \eta[\![e]\!]^{-1}$$

An analogous derivation is also required for polymorphic application and $\texttt{App}\ e$ expansion in call-by-value, which both follow from the $\beta_{\forall}$ axiom. □

Corollary 6 (Equational Correspondence)**.** *There is an equational correspondence between both call-by-name and call-by-value System F and the corresponding sublanguage of $\mathcal{IL}$. Namely, the following properties hold:*

(1) If $\Gamma \vdash e_1 = e_2 : \tau$ in System F, then $[\![\Gamma]\!] \vdash [\![e_1]\!] = [\![e_2]\!] : [\![\tau]\!]$ in $\mathcal{IL}$.
(2) If $\Gamma \vdash e_1 = e_2 : \sigma$ in $\mathcal{IL}$, then $[\![\Gamma]\!]^{-1} \vdash [\![e_1]\!]^{-1} = [\![e_2]\!]^{-1} : [\![\sigma]\!]^{-1}$ in System F.
(3) For all $\Gamma \vdash e : \tau$ in System F, $\Gamma \vdash [\![[\![e]\!]]\!]^{-1} = e : \tau$.
(4) For all $\Gamma \vdash e : \sigma$ in $\mathcal{IL}$, $\Gamma \vdash [\![[\![e]\!]^{-1}]\!] = e : \sigma$.

Corollary 7 (Soundness and Completeness)**.** *For any $\Gamma \vdash e_i : \tau$ in either call-by-name or call-by-value System F, $\Gamma \vdash e_1 = e_2 : \tau$ if and only if $[\![\Gamma]\!] \vdash [\![e_1]\!] = [\![e_2]\!] : [\![\tau]\!]$.*