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 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 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.

Draft March 2020. We would appreciate any comments or feedback to the email addresses below.

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

1 INTRODUCTION

Polymorphism supports re-use by allowing one chunk of executable to work on 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 re-use if we could specialize to, say, "any type that is 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*:

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

We would like to be able to control 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 the type system.

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 (Section 2):

- *Representation.* How is this argument represented at runtime?
- *Levity.* What is the evaluation strategy of this argument (call-by-value or call-by-need)?
- *Calling convention.* For functions, how many arguments does this function take before its code can be executed (its arity)?

Indeed, it turns out that 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 simple, uniform language, but the compiler needs a more expressive intermediate language so that it can express low-level representation choices, and expose that code to the optimizer. For example, the programmer might work exclusively with boxed integer values, of type `Int` say, but the intermediate language

Authors' addresses: Paul Downen; Zena M. Ariola, University of Oregon, 1477 E. 13th Ave. Eugene, Oregon, 97403, USA, pdownen@cs.uoregon.edu, ariola@cs.uoregon.edu; Simon Peyton Jones, Microsoft Research, 21 Station Rd. Cambridge, CB1 2FB, UK, simonpj@microsoft.com; Richard A. Eisenberg, Bryn Mawr College, 101 N. Merion Ave. Bryn Mawr, PA, 19010, USA , Tweag I/O, Cambridge, UK, rae@richarde.dev.

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].

In this paper we build directly on several earlier works that track different representations [Eisenberg and Peyton Jones 2017] and function arities [Downen et al. 2019] within a type and kind system, but here we bring them together into a single framework, more powerful and more precise than any of its predecessors. Specifically, we make these contributions:

- We introduce a polymorphic intermediate language that statically captures *calling conventions in kinds* (Section 3), and has polymorphism over the *representation*, *levity*, and *arity* of types.
- We show how to compile our polymorphic intermediate language to a more conventional lower-level representation (Section 4) that has different representations of values (*e.g.,* pointers versus integers) and multi-arity functions, but is *not* polymorphic. Crucially, compilation is driven by kinds and uses static restrictions to ensure that polymorphic code can always be compiled to monomorphic code, without sacrificing type erasure or duplicating code.
- Even though we statically track calling conventions in kinds, there might still be opportunities for improving a call at runtime. We describe a small extension to our intermediate language to allow for dynamic checks on the arities of closures at runtime, so that we can use the best arity available during execution (Section 5).
- 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 6).
- We discuss how having kinds as calling conventions can be used to generalize user-defined data types (Section 7) for expressing more efficient code in a high-level language and enriching a source language with both lazy and eager data types.
- We provide evidence of correctness for the full compilation process (Theorems 2 and 4) for call-by-name and call-by-value evaluation, thus showing that our intermediate language is equally well-suited for representing *both eager and lazy* functional languages.

We discuss related work and conclude in Sections 8 and 9.

2 SYSTEM F IS IMPRACTICAL AS AN INTERMEDIATE LANGUAGE

When developing an intermediate language for functional programs, System F serves as a wonderful starting point that solves the challenge of polymorphism. However, it does not address several concerns involving the efficiency of those programs. Imagine a baseline of System F: everything is represented uniformly by a pointer to support polymorphism; function arguments are passed one-at-a-time to support currying; and the evaluation strategy is fixed as call-by-value or call-by-name (the particular choice is not important) to support either laziness or strictness, but not both. Before going into our main contribution—a type and kind system that enables efficient calling conventions—in the next section, let us unpack these individual issues in more detail.

2.1 Polymorphism

Many optimizing compilers use *type erasure* to handle polymorphic functions, wherein typing information is not present at runtime. Consider the lowly identity function $id : t \to t$. Its type tells us that id works over *any* type t. Yet even a function as simple as id must do *some* work. It must know where to find its argument, where to put its result, and then copy from the first spot to the second. Type erasure imposes the constraint that the code generated for id must be able to do all of this for *any* type t: type erasure means that we cannot inspect the particular choice of t, so we must have one sequence of instructions that works for any t.

Many types are represented by pointers, suggesting that we can compile *id* to expect one pointer as input and produce one pointer as output. Unfortunately, this simplification prevents specializing *id* to work with other types of runtime values like `Int#`—the type describing machine integers—which may have a different size or use different registers than pointers.

We thus arrive at a dilemma: either somehow restrict *id* not to be able to work with `Int#`, or ban types like `Int#` from our language altogether. We say that the latter is impractical, because constant, unavoidable indirection via pointers is too slow for industrial programming. Effectively, a naïve implementation of System F makes this latter, impractical choice. Instead, a real-world programming language must somehow restrict *id*. We do so by classifying `Int#` as having a different *kind* than types represented by pointers [Eisenberg and Peyton Jones 2017].

This second, more practical, choice is similar to the move to System F_ω, an extension of System F with many different kinds of types for expressing type operators like `Array`. In System F_ω, we are forced to state the kind of types a type variable like t ranges over. For example, the identity function is elaborated as $\forall t{:}\star.t \to t$, where the kind $\star$ must somehow specify that t is represented as a pointer. A type like `Int#`, then, must have a kind which is different from $\star$.

2.2 Higher-order functions

Functional programming languages get significant mileage out of *currying*: a function *plus* of type `Int` → `Int` → `Int` is regarded as a function of one argument (of type `Int`) that returns a function of one argument (another `Int`), that finally returns an `Int`. Such a function can be *partially applied*. For example *map* (*plus* 1) is a function that increments every element in a list; here both *map* and *plus* are partially applied. Currying is very convenient for programmers, and can lead to measurable reduction in code size [Arvind and Ekanadham 1988].

But we do not want to *implement* currying by making every multi-argument function return a chain of (heap-allocated) function closures; that would be unacceptably inefficient. So compilers like GHC and OCamlC go to great lengths to group together adjacent lambdas, and pass the arguments "all at once" to an uncurried variant of the function [Leroy 1990; Marlow and Peyton Jones 2004]. The number of arguments simultaneously expected by a function before doing work is called its *arity*. When calling a (let-bound) function whose definition is in scope, GHC can ensure that the right number of arguments are passed to match its arity and thus producing efficient code.

But calling unknown (lambda-bound) functions whose definitions cannot be found is harder. Consider the third clause of *zipWith*:

$$\begin{array}{ll} \mathit{zipWith}\ f\ []\ \mathit{ys} & = [] \\ \mathit{zipWith}\ f\ \mathit{xs}\ [] & = [] \\ \mathit{zipWith}\ f\ (x : \mathit{xs}')\ (y : \mathit{ys}') & = (f\ x\ y) : (\mathit{zipWith}\ \mathit{xs}'\ \mathit{ys}') \end{array}$$

The arity of the function f passed to *zipWith* might be 1 or 2. It might even be 3 or 0 (if it is an unevaluated thunk), depending on the caller. So the code for *zipWith* is forced to account for all these possibilities in the call $(f\ x\ y)$. But we could generate *much* better code if we statically knew f's arity and, building on earlier work [Downen et al. 2019], that is what we do here.

2.3 Evaluation strategy

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 *later, on-demand*? The designer of a particular version of System F makes this choice in its semantics. Real-world programming languages also have committed to a choice here. This choice is frequently 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, for example, to make a control-flow

$$\tau, \sigma \in \mathit{Type} ::= t \mid T_p \mid T_d \mid \tau \rightsquigarrow \sigma \mid \forall \chi.\sigma \mid {}^{\gamma}\{\tau\} \qquad \chi \in \mathit{TyVar} ::= t{:}\kappa \mid g \mid r \mid v$$
$$\kappa \in \mathit{Kind} ::= \mathtt{TYPE}\ \rho\ \nu \qquad T_d \in \mathit{DataType} ::= \mathtt{Int}^{\gamma} \mid \ldots$$
$$\rho \in \mathit{Representation} ::= r \mid \mathtt{PtrR} \mid \mathtt{IntR} \mid \ldots \qquad T_p \in \mathit{PrimType} ::= \mathtt{Int\#} \mid \ldots$$
$$\gamma \in \mathit{Levity} ::= g \mid \mathtt{L} \mid \mathtt{U}$$
$$\nu \in \mathit{Convention} ::= v \mid \mathtt{Eval}^{\gamma} \mid \mathtt{Call}[\alpha] \qquad \alpha \in \mathit{Arity} ::= \varepsilon \mid \rho, \alpha \mid \mathit{arity}(\nu)$$
$$e \in \mathit{Expr} ::= x \mid c \mid \mathtt{I\#}^{\gamma}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathtt{I\#}\ x \longrightarrow e' \mid \lambda x{:}\tau.e \mid e\ e \mid \lambda\chi.e \mid e\ \phi \mid \mathtt{Clos}^{\gamma}\ e \mid \mathtt{App}\ e$$
$$c \in \mathit{Const} ::= n \mid \mathit{op} \qquad \mathit{op} \in \mathit{PrimOp} ::= \mathtt{error} \mid \ldots$$
$$P \in \mathit{Passive} ::= x \mid c \mid \mathtt{I\#}^{\gamma}\ P \mid \mathtt{Clos}^{\gamma}\ e \mid \lambda\chi.P \mid P\ \phi \qquad \phi \in \mathit{Erasable} ::= \tau \mid \gamma \mid \rho \mid \nu$$

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

decision, and only once. However, regardless of a language's choice of evaluation strategy, some scenarios demand making the opposite choice. Programmers in an eager language sometimes use constructs like *Delay* and *Force* [Wadler et al. 1998] to embed a lazy evaluation, and programmers in a lazy language use strict bindings or primitives (such as Haskell's *seq*) to force eager evaluation. We thus want a language that supports both lazy and eager evaluation, giving the programmer a convenient means to choose between the two.

With the terminology of "liftedness"—whether a variable of a type can be bound to a computation, or only to values—the types of an eager language (like OCaml) are all unlifted, whereas the types of a lazy language (like Haskell) are all lifted. We can thus call the change between lazy and eager computation a change of liftedness, or *levity*. A function that can work with both lifted and unlifted types is *levity polymorphic*.[1] In order to support levity polymorphism in our language, we once again leverage the kind system, distinguishing between the two modes in the kinds of types.

3 OUR KEY CONTRIBUTION: THE INTERMEDIATE LANGUAGE ($\mathcal{IL}$)

Previous research has suggested that *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 that combines several different previous works that focused on addressing representations [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.

Why *kinds*, rather than *types*? Because of polymorphism. A type can tell us important runtime details (such as the representations of values or the arity of functions), but polymorphism means that types might be statically unknown inside of a definition. Yet we still want to compile polymorphic definitions, and in such a way that the same code can be reused for every instantiation. Therefore, we encode just enough intensional information within kinds to express the low-level details needed to compile polymorphic code. Think of these kinds as a coarser-grained descriptions, that leave more subtle issues like safety to the finer-grain world of types. Better still, making details like

[1] Previous work [Eisenberg and Peyton Jones 2017] is titled *Levity Polymorphism*. Yet the paper does not, in our opinion, deliver exactly on the promise in the title. Instead, it describes *representation polymorphism*, choosing not to distinguish between levity and runtime representation. As a consequence, levity polymorphism is only possible (and in fact, mandatory) as part of representation polymorphism.

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

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau}\ \textsc{var} \qquad \frac{c : \tau}{\Gamma \vdash c : \tau}$$

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

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

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow \sigma \quad \Gamma \vdash P : \tau \quad \Gamma \vdash \tau\ \textbf{mono-rep}}{\Gamma \vdash e\ P : \sigma}\ \textsc{lam-p-E}$$

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

$$\frac{\Gamma, \chi \vdash e : \sigma \quad \Gamma \vdash \forall \chi.\sigma : \kappa}{\Gamma \vdash \lambda \chi.e : \forall \chi.\sigma}\ \forall_I \qquad \frac{\Gamma \vdash e : \forall \chi.\sigma \quad \Gamma \vdash [\phi/\chi]\ \textbf{poly}}{\Gamma \vdash e\ \phi : \sigma[\phi/\chi]}\ \forall_E$$

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

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

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

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

Fig. 2. Type system of $\mathcal{IL}$

representation, arity, and levity explicit lets us express when they do *not* need to be known—analogous to polymorphic code that is independent of a type—thereby improving code reuse.

The goal of this paper is thus to provide a type system in which the kind κ of a type $\tau{:}\kappa$ classifies both the *representation* and the *calling convention* of values of type τ. To give ourselves a solid foundation, we focus on the compiler's *explicitly-typed intermediate language*, rather than the source language. Our intermediate language, which we call $\mathcal{IL}$, is an explicitly-typed λ-calculus based closely on System F, so much of its syntax (Fig. 1) and typing rules (Figs. 2 and 3) should be familiar. In particular the syntax for expressions e includes:

- Variables x, constants c.
- Two types for integer values, $\texttt{Int}^{\gamma}$ (boxed, where γ refers to the levity, see Section 3.1) and `Int#` (unboxed), together with expression forms for explicitly boxing ($\texttt{I\#}^{\gamma}\ e$) and unboxing (**case**); see Section 3.2.
- Functions whose arity is statically known, introduced and eliminated with the familiar looking forms $\lambda x{:}\tau.e$ and $e_1\ e_2$. In spite of apparent familiarity, the notational change in the type to a wavy arrow ($\tau_1 \rightsquigarrow \tau_2$), rather than the more conventional ($\tau_1 \to \tau_2$), signals some rather subtle, but crucial, semantic differences that we detail in Section 3.3.
- Functions of unknown arity, introduced and eliminated with the new forms $\texttt{Clos}^{\gamma}\ e$ and `App` e, respectively, are also detailed in Section 3.3.

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

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

$$\frac{\Gamma \vdash \gamma \ \mathbf{lev} \quad \Gamma \vdash \tau : \texttt{TYPE}\ \rho\ \nu}{\Gamma \vdash {}^{\gamma}\{\tau\} : \texttt{TYPE PtrR Eval}^{\gamma}}\ \textsc{Tcl} \qquad \frac{\Gamma, \chi \vdash \sigma : \texttt{TYPE}\ \rho\ \nu \quad \Gamma \vdash \texttt{TYPE}\ \rho\ \nu\ \mathbf{kind}}{\Gamma \vdash \forall \chi.\sigma : \texttt{TYPE}\ \rho\ \nu}\ \textsc{forall}$$

$$\frac{\Gamma \vdash \tau_1 : \texttt{TYPE}\ \rho_1\ \nu_1 \quad \Gamma \vdash \tau_2 : \texttt{TYPE}\ \rho_2\ \nu_2}{\Gamma \vdash \tau_1 \rightsquigarrow \tau_2 : \texttt{TYPE PtrR Call}[\rho_1, \mathit{arity}(\nu_2)]}\ \textsc{arrow} \qquad \frac{\Gamma \vdash \tau : \kappa \quad \kappa = \kappa'}{\Gamma \vdash \tau : \kappa'}\ \textsc{k-conv}$$

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

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

Monomorphism restrictions:

$$\frac{\Gamma \vdash \tau : \texttt{TYPE}\ \rho\ \nu \quad \vdash \rho\ \mathbf{rep}}{\Gamma \vdash \tau\ \mathbf{mono\text{-}rep}} \qquad \frac{\Gamma \vdash \tau : \texttt{TYPE}\ \rho\ \nu \quad \vdash \nu\ \mathbf{conv}}{\Gamma \vdash \tau\ \mathbf{mono\text{-}conv}}$$

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

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

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

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

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

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

Fig. 3. Kind and levity system of $\mathcal{IL}$

- Type abstraction $\lambda\chi.e$ and application $e\ \phi$. The only unusual thing here is that binders χ and arguments ϕ range over four different sorts of (erasable) variables and arguments respectively. We use a consistent naming convention, with a Latin-font name for variables (t, g, v, and r) and the corresponding Greek-font name for the syntactic category (τ, γ, ν, and ρ respectively).

Note that a subset of these expressions are *Passive* (denoted by P), meaning that they do not require evaluation when passing them to a function or binding them to a variable. Of note, this classification accounts for eventual *type erasure* (which is why $\lambda\chi.e$ is passive only when e is and $P\ \phi$ is passive because ϕ arguments do not survive compilation). Passive expressions include *closures* $\texttt{Clos}^{\gamma}\ e$ as usual, but not *functions* $\lambda x{:}\tau.e$ because functions are *called* instead of *evaluated*. Types τ, σ 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 $\forall\chi.\sigma$, function types $\tau \rightsquigarrow \sigma$ and a closure type ${}^{\gamma}\{\tau\}$.

3.1 Kinds, representations, calling conventions, and levities

A kind κ has the form `TYPE` ρ ν, where ρ describes the *representation* of the type, and ν describes its *calling convention*, that is, what operations are allowed on that type. We do not support higher kinds, nor kind polymorphism; adding either is entirely straightforward, but introduces distracting details. Here we focus on the essentials of the intensional details in kinds.

Suppose $x : \tau : \texttt{TYPE}\ \rho\ \nu$; that is, x is a term variable of type τ, whose kind is `TYPE` ρ ν. We now describe what ρ and ν mean. The *representation* ρ describes how the value of x is represented at runtime. Referring to Fig. 1, ρ can be:

- `PtrR`, meaning that x is represented by a pointer into the garbage-collected heap.
- `IntR`, meaning that x is represented by a machine integer (not a pointer). In reality we would have many such kinds, for integers of different widths, for floating point values, and so on.
- r, a representation variable, which can be bound by a $\forall$; that is, we support representation polymorphism [Eisenberg and Peyton Jones 2017].

The *convention* ν describes how x may be consumed. Referring to Fig. 1, we see that ν can be:

- $\texttt{Eval}^{\texttt{U}}$, 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.
- $\texttt{Eval}^{\texttt{L}}$, 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.
- $\texttt{Eval}^{g}$, where g is a levity variable; that is, we support levity polymorphism.
- `Call`$[\alpha]$, meaning that x is a function (not a thunk) with an *arity* described by α. The arity of a function is either a fixed list $\rho_1, \ldots, \rho_n$, in which case x takes precisely n arguments, whose representations are given by ρ_i. Otherwise, the arity will have the form $\rho_1, \ldots, \rho_n, \mathit{arity}(\nu)$, meaning that x takes *at least* n arguments, followed by possibly some more arguments given by the arity of another calling convention ν. `Call` is a completely new concept: see Section 3.4.
- v, a calling-convention variable, which can be bound by a $\forall$; that is, we support polymorphism over calling conventions.

The kind `TYPE PtrR` $\texttt{Eval}^{\texttt{L}}$ expresses the uniform representation of a value in a lazy language, as a pointer to a lifted (i.e. possibly a thunk) object. Because this kind is so common, 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 `TYPE PtrR` $\texttt{Eval}^{\texttt{U}}$ and $\texttt{Eval}^{\texttt{L}}$ would be used sparingly, if at all.

3.2 Boxed and unboxed data types

Nearly thirty years ago, GHC introduced the idea of distinguishing boxed and unboxed data types in its intermediate language [Peyton Jones and Launchbury 1991]. We adopt this idea, but as a special case in a more general framework. Specifically, we have:

- A primitive type `Int#`, of kind `TYPE IntR` $\texttt{Eval}^{\texttt{U}}$. The `IntR` says that `Int#` is represented by a machine integer; while the $\texttt{Eval}^{\texttt{U}}$ says that it is unlifted (cannot be a thunk). The type `Int#` comes with literal constants n, and primitive operations op over it. We specify just one such primitive type, but in reality there would be many more, with different representations (machine integers of various widths, floating point numbers of various widths, etc).
- A boxed type $\texttt{Int}^{\gamma}$, of kind `TYPE PtrR` $\texttt{Eval}^{\gamma}$. This kind specifies that an $\texttt{Int}^{\gamma}$ is represented by a pointer to a heap-allocated object (hence "boxed"), and that it may be lifted (if γ is `L`) or unlifted (if γ is `U`).

In reality there would be a way to declare new user-defined algebraic data types, and these types need not be levity polymorphic (i.e., have a γ argument). The sole goal of distinguishing two types is efficiency. If we had only boxed integers, then even simple addition would be forced to evaluate

and unbox each argument, and box up the result. By making these operations explicit we expose much more to the optimizer, and can eliminate lots of intermediate boxes. For example, consider:[2]

$$\begin{array}{ll} \mathit{plus} : \texttt{Int} \rightsquigarrow \texttt{Int} \rightsquigarrow \texttt{Int} & \mathit{sumFrom} \quad\quad : \texttt{Int} \rightsquigarrow \texttt{Int} \\ \mathit{plus}\ (\texttt{I\#}\ x)\ (\texttt{I\#}\ y) = \texttt{I\#}(\texttt{plus\#}\ x\ y) & \mathit{sumFrom}\ (\texttt{I\#}\ 0) = \texttt{I\#}\ 0 \\ & \mathit{sumFrom}\ (\texttt{I\#}\ n) = \mathit{plus}\ (\texttt{I\#}\ n)\ (\mathit{sumFrom}\ (\texttt{I\#}(\texttt{minus\#}\ n\ 1)) \end{array}$$

where `plus#` and `minus#` are primitive operations (extending *PrimOp* from Fig. 1), both of type $\texttt{Int\#} \rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#}$.

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

$$\begin{array}{ll} \mathit{sumFrom} \quad\quad : \texttt{Int} \rightsquigarrow \texttt{Int} & \mathit{sumFrom\#} \quad : \texttt{Int\#} \rightsquigarrow \texttt{Int\#} \\ \mathit{sumFrom}\ (\texttt{I\#}\ n) = \texttt{I\#}\ (\mathit{sumFrom\#}\ n) & \mathit{sumFrom\#}\ 0 = 0 \\ & \mathit{sumFrom\#}\ n = \texttt{plus\#}\ n\ (\mathit{sumFrom\#}\ (\mathit{minus\#}\ n\ 1)) \end{array}$$

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 allocation. *sumFrom* becomes just a wrapper around *sumFrom#* which handles all of the issues of boxing and unboxing. Instead of allocating several extraneous boxes at each step of the loop as before, the optimized code will only ever unbox the given number once at the start and then allocate the final box at the end. This is a huge gain in both time and space!

We add one new refinement, however: we can distinguish $\texttt{Int}^{\texttt{L}}$ from $\texttt{Int}^{\texttt{U}}$, thus reusing the same function—and eventually generating the same code—for both strict and lazy languages. We will return to this point after having introduced levity polymorphism (see section Section 3.5).

3.3 Lambdas and closures

Consider these two function definitions in, say, OCaml:

$$f_1 = \lambda x.\, \lambda y.\ \textbf{let}\ z = \texttt{print}\ x\ \textbf{in}\ \texttt{plus\#}\ x\ y \qquad f_2 = \lambda x.\ \textbf{let}\ z = \texttt{print}\ x\ \textbf{in}\ \lambda y.\, \texttt{plus\#}\ x\ y$$

These two functions are η-equivalent to one another, and yet, they are not the same! The side effect of printing x reveals the difference when the functions are partially applied. For example, $\textbf{let}\ z = f_1\ 10\ \textbf{in}\ 20$ will just return 20, but $\textbf{let}\ z = f_2\ 10\ \textbf{in}\ 20$ will first print 10 and then return 20. For this reason, the OCaml compiler cannot freely η-convert between f_1 and f_2, due to side effects and eagerness changing the result of a partial application.

Even in a pure, lazy language like Haskell, a similar issue arises once we think about operational concerns. Consider now these two Haskell function definitions [Downen et al. 2019]:

$$\begin{array}{l} f_1, f_2 : \texttt{Int\#} \to \texttt{Int\#} \to \texttt{Int\#} \\ f_1 = \lambda x.\, \lambda y.\ \textbf{let}\ z\ =\ h\ x\ x\ \textbf{in}\ e\ y\ z \\ f_2 = \lambda x.\ \textbf{let}\ z\ =\ h\ x\ x\ \textbf{in}\ \lambda y.\, e\ y\ z \end{array}$$

Suppose we have a call $(f_1\ 10\ 20)$. Currying suggests that we should first call $(f_1\ 10)$, returning a (heap-allocated) function closure, and then call that closure with 20. This is unacceptably inefficient in practice; instead, we want to pass 10 and 20 *simultaneously*, with no intermediate function closure. Doing so is fine, because, f_1 can do no work until it is applied to two arguments.

But f_2 is quite different: we *should* pass the arguments one at a time. Imagine we had the call $\mathit{map}\ (f_2\ 10)\ \mathit{xs}$: then we want to share the computation of $(h\ 10\ 10)$ among all the elements of xs. Unlike f_1, function f_2 can do (potentially expensive) work when applied to one argument.

[2] Here and elsewhere we reduce notational clutter by using syntactic sugar such as pattern matching.

In this case, we could η-expand f_2 anyway, to get f_1. Now calls to f_2 would be more efficient—but at the cost of an asymptotic slowdown when used with *map*. We also saw previously that if we have side effects, like in OCaml, then η-expansion changes *semantics*, not just *efficiency*. Even Haskell has the *seq* operator, which also makes η-expansion semantically unsound. In short, unrestricted η-expansion is not allowed in any realistic source language.

In Haskell the distinction between f_1 and f_2 is not apparent in their types; indeed they both have the same type. The idea of Downen et al. [2019] is to express arity (*i.e.,* the ability to do computationally-safe η-expansion) in the types. In particular, in $\mathcal{IL}$ *any expression e whatsoever of type* $\tau_1 \rightsquigarrow \tau_2$ *can safely be* η*-expanded to* $\lambda x.\, e\ x$.

What, then, of currying? If f_2 is eta-expandable to f_1 without changing the computational cost, how can we achieve the sharing desired by the Haskell programmer above? We can do it like this:

$$f_3 :: \texttt{Int\#} \rightsquigarrow {}^{\texttt{L}}\{\texttt{Int\#} \rightsquigarrow \texttt{Int\#}\}$$
$$f_3 = \lambda x.\ \textbf{let}\ z\ =\ h\ x\ x\ \textbf{in}\ \texttt{Clos}^{\texttt{L}}\ (\lambda y.\, e\ y\ z)$$

Here the "return a function closure" part of currying has become fully explicit in the expression $\texttt{Clos}^{\gamma}\ e$, and is reflected in f_3's type by ${}^{\texttt{L}}\{\texttt{Int\#} \rightsquigarrow \texttt{Int\#}\}$.

How can f_3 be called? Clearly we cannot write (f_3 10 20) because the application ($e_1\ e_2$) requires that $e_1 : \tau_1 \rightsquigarrow \tau_2$. Instead, `Clos` comes with its dual elimination form, `App`, and we call f_3 by writing (`App` (f_3 10) 20). This call directly expresses the idea of applying f_3 to one argument 10, and then applying the resulting function closure to the second argument 20. The typing rules for `Clos` and `App` are just as you would expect: see rules Clo-I and Clo-E in Fig. 2.

3.4 Kinds are calling conventions

We motivated the idea that the *type* of a function could express its *arity*; just count the arrows. For example, $\texttt{Int\#} \rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#}$ is the type of an arity 2 function. But suppose we have

$$id : \forall(t{:}\texttt{TYPE PtrR}\ \nu).\ t \rightsquigarrow t$$

What is *id*'s arity? You might reasonably answer "one," but what about when t is instantiated to ($\texttt{Int\#} \rightsquigarrow \texttt{Int\#}$)? That type application presumably has type ($\texttt{Int\#} \rightsquigarrow \texttt{Int\#}$) $\rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#}$, which appears to have arity 2. Suddenly it is not clear how many arguments *id* expects.

One way to address this problem, used by [Downen et al. 2019], is to *use the kind system to prevent instantiating type variable with an arrow type*. But that is a very drastic restriction: it means you cannot have, say, a list of functions; instead you would be forced to wrap them in `Clos`.

We take a more powerful approach here: instead of counting the arrows in the function's type (which sometimes fails in the presence of type polymorphism), we simply look at the type's kind. That kind expresses the function's arity (and argument representations). For example:

$$\begin{aligned} \texttt{Int\#} \rightsquigarrow \texttt{Int\#} &\ :\ \texttt{TYPE PtrR Call[IntR]} \\ \texttt{Int\#} \rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#} &\ :\ \texttt{TYPE PtrR Call[IntR, IntR]} \\ \texttt{Int\#} \rightsquigarrow \texttt{Int}^{\texttt{L}} \rightsquigarrow \texttt{Int\#} &\ :\ \texttt{TYPE PtrR Call[IntR, PtrR]} \end{aligned}$$

The calling-convention component in these kinds is $\texttt{Call}[\alpha]$, where the arity list α describes the number and representation of arguments expected by the function. We need the latter so that, during compilation (Section 4), we can eta-expand a function to its full arity.

The kinding rule for ($\tau_1 \rightsquigarrow \tau_2$) must track these arities. Looking at rule Arrow in Fig. 3 we see that the kind of ($\tau_1 \rightsquigarrow \tau_2$) has a calling convention of $\texttt{Call}[\rho_1, \mathit{arity}(\nu_2)]$. The first argument has representation ρ_1, the representation of τ_1. The rest of the arguments come from the calling convention of τ_2, via the type-level function *arity*, whose behavior is defined in Fig. 3. It returns the arity of $\texttt{Call}[\alpha]$; and the empty arity in the case of $\texttt{Eval}^{\gamma}$ since no arguments are needed

to evaluate a non-function. But v_2 might also be a variable v, and then $arity(v)$ is stuck; that is why $arity(v)$ is part of the syntax of α in Fig. 1. Rule K-CONV allows calls to *arity* to be calculated whenever desired.[3]

Here at last we see the key contribution of this paper: the calling convention of a function is expressed in the kind of its type, rather than in the type itself. What forces us to use kinds rather than types? Answer: the desire for abstraction, and specifically polymorphism. In a monomorphic system, "types are calling conventions" works just fine.

3.5 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, consider the `error` function—a motivating example of [Eisenberg and Peyton Jones 2017]—which takes an error code of type `Int#` and halts the program with some message. It can be given this type:

$$error : \forall r\, g.\ \forall (t : \mathtt{TYPE}\ r\ \mathtt{Eval}^{g}).\ \mathtt{Int\#} \rightsquigarrow t$$

Here, *error* is polymorphic in both the representation r and the levity g of the returned value because, in fact, it never returns a value. Lacking this polymorphism, we would be forced to define a whole family of monomorphic versions of the *error* function, one for each return representation and levity, which would be extremely painful—especially since they compile to the same code.

Using polymorphism over levity also allows us to write some functions that work uniformly over both strict and lazy values. For example, adding two boxed integers can be defined thus

$$\begin{aligned} plus\ &:\ \forall g_1.\, \forall g_2.\, \forall g_3.\, \mathtt{Int}^{g_1} \rightsquigarrow \mathtt{Int}^{g_2} \rightsquigarrow \mathtt{Int}^{g_3} \\ plus\ g\ (\mathtt{I\#}\ x)\ (\mathtt{I\#}\ y) &= \mathtt{I\#}^{g_3}\ (\mathtt{plus\#}\ x\ y) \end{aligned}$$

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

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

Notice that in this definition, the levities g_i of the argument and return types $\mathtt{Int}^{g_i}$ are statically unknown, so we must be able to pattern-match on and return values with unknown levities. Specifically, rule CASE in Fig. 2 allows a case-expression to scrutinize an integer of arbitrary levity γ. Operationally, in a polymorphic situation the case-expression has to test the scrutinee to see if it is a thunk (in case the 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 case for handling a thunk is simply dead code.[4]

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

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

From a representation point of view, an $\mathtt{Array\#}^{\gamma}$ 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 $\mathtt{Array\#}^{\mathsf{L}}\ (\mathtt{Array\#}^{\mathsf{U}}\ \mathtt{Int}^{\mathsf{U}})$ is a lifted array of pointers, each of which points directly

[3] Alternatively, we could require $arity(v)$ to be fully calculated in the ARROW kinding rule. This would let us remove $arity(v)$ from the grammar of arities, but also forces an additional restriction on the formation of types and expressions, specifically ARROW and LAM-I, to rule out $arity(v)$. The cost of such a restriction is to break the existing property that, except for the $\forall$ quantifier, any type made from well-kinded types is itself well-kinded.

[4] We assume here that the concrete, run-time representation of evaluated lifted values is the same as the representation of unlifted values. This is true in GHC and seems likely in other systems that support laziness, but it is conceivably an invalid assumption in some systems.

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].

3.6 Polymorphism in calling convention

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

$$revapp\ x\ f = f\ x$$

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

(1) $revapp : \forall(t{:}\texttt{TYPE PtrR Eval}^{\texttt{L}}).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#}$
(2) $revapp : \forall(t{:}\texttt{TYPE IntR Eval}^{\texttt{U}}).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#}$

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[5]. So we can choose (1) or (2), but not both.

On the other hand, consider these other possible types:

(3) $revapp : \forall(t{:}\texttt{TYPE PtrR Eval}^{\texttt{U}}).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#}$
(4) $revapp : \forall(t{:}\texttt{TYPE PtrR Call[IntR]}).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#}$

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 v, thus:

(4) $revapp : \forall v.\forall(t{:}\texttt{TYPE PtrR } v).\ t \rightsquigarrow (t \rightsquigarrow \texttt{Int\#}) \rightsquigarrow \texttt{Int\#}$

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), so we can be completely polymorphic in its representation, thus:

(5) $revapp : \forall v\ r\ g.\ \forall(t_1{:}\texttt{TYPE PtrR } v)\ (t_2 : \texttt{TYPE } r\ \texttt{Eval}^{g}).\ t_1 \rightsquigarrow (t_1 \rightsquigarrow t_2) \rightsquigarrow t_2$

But notice that, unlike the argument type t_1, *revapp* cannot be polymorphic in the calling convention of t_2, as we discussed in Section 3.4. It can, however, be evaluated with any levity.

3.7 Restrictions on polymorphism

Of course we have the usual restrictions on polymorphism,[6] but the polymorphism $\mathcal{IL}$ introduces some new issues. We have already seen how unrestricted polymorphism is incompatible with efficient static code generation[7] in Section 3.6, where we cannot allow *revapp*'s type argument t_1 to have a representation-polymorphic kind. A second restriction is demonstrated by this function:

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

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

(1) $twice : \forall(t{:}\texttt{TYPE PtrR Eval}^{\texttt{L}}).\ (t \rightsquigarrow t) \rightsquigarrow t \rightsquigarrow t$
(2) $twice : \forall(t{:}\texttt{TYPE PtrR Eval}^{\texttt{U}}).\ (t \rightsquigarrow t) \rightsquigarrow t \rightsquigarrow t$

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

[5] 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.
[6] For example, every free variable that appears anywhere in the type checking judgment $\Gamma \vdash e : \tau$ must be bound by Γ.
[7] 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 a static code generation.

The restrictions on polymorphism in our language are embodied in the shaded premises in the type system of Fig. 2. The judgment $\Gamma \vdash \tau$ **mono-rep**, defined in Fig. 3 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. 2 we see the shaded premises:

- Rule LAM-E: for a general application, the argument type must be monomorphic in both the representation (so that we know how to pass it to the function), and convention (so that we know when to evaluate it, *or* for first-class functions as arguments, how to define it).
- Rule CLO-I: this rule constructs a closure of an unknown arity from an expression with a known arity (as we elaborated in Section 3.3). As such, it needs to know the arity of the contained expression in order to define the closure.

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 4, how to compile $\mathcal{IL}$ into a lower level "machine language" $\mathcal{ML}$. In the translation from $\mathcal{IL}$ to $\mathcal{ML}$ in Fig. 6, we need exactly the shaded monomorphic restrictions of Fig. 2. If any of these restrictions were removed, then there would be expressions that are well-typed, yet un-compilable.

Our rules also include two additional, unshaded, monomorphism restrictions, in the LAM-I and LAM-P-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 4 *does not* require any monomorphism restrictions in LAM-I and LAM-P-E: they could be deleted and yet all closed, well-typed expressions could still be compiled. This example shows how 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.

3.8 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. 3). Intuitively, this is because these quantifiers will be totally *erased* by compilation, and have no impact on the final run-time code. Since kinds are meant to reflect the representations and actions that occur during run-time, the $\forall$ is "invisible" to the lower-level machine.

However, now that variables may appear in kinds, we must be careful to avoid such a variable escaping its scope. For example, the following type is not well-kinded:

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

Here the representation r of the first parameter escapes in the calling convention of this 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. 3 and the second premise of rule $\forall_I$ in Fig. 2.

3.9 Let bindings

$\mathcal{IL}$ does not have a **let**-binding construct but, as usual, a non-recursive **let** can be regarded as shorthand for lambda and application: $\mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ e' \triangleq (\lambda x{:}\tau.\, e')\, e$. The typing rule follows by composing LAM-I and LAM-E:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash e' : \sigma \quad \Gamma \vdash \tau\ \mathbf{mono\text{-}rep} \quad \Gamma \vdash \tau\ \mathbf{mono\text{-}conv}}{\Gamma \vdash \mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ e' : \sigma}\ \text{LET}$$

Type-based passable expressions (*i.e.*, substitutable values): $\boxed{\Gamma \vdash e\ \textbf{pass}}$

$$\frac{}{\Gamma \vdash P\ \textbf{pass}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \mathtt{TYPE}\ \rho\ \mathtt{Eval}^{\mathsf{L}}}{\Gamma \vdash e\ \textbf{pass}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \mathtt{TYPE}\ \rho\ \mathtt{Call}[\alpha]}{\Gamma \vdash e\ \textbf{pass}}$$

Equational axioms (in each rule, assume that $\Gamma \vdash S\ \textbf{pass}$):

$$(\beta_{\rightsquigarrow}) \quad (\lambda x{:}\tau.e)\ S = e[S/x] \qquad (\eta_{\rightsquigarrow}) \quad \lambda x{:}\tau.(e\ x) = e : \tau \rightsquigarrow \sigma$$

$$(\beta_{\forall}) \quad (\lambda \chi.e)\ \phi = e[\phi/\chi] \qquad (\eta_{\forall}) \quad \lambda \chi.(e\ \chi) = e : \forall \chi.\sigma$$

$$(\beta_{\{\}}) \quad \mathtt{App}\ (\mathtt{Clos}^{\gamma}\ e) = e \qquad (\eta_{\{\}}) \quad \mathtt{Clos}^{\gamma}\ (\mathtt{App}\ S) = S : {}^{\gamma}\{\sigma\}$$

$$(\beta_{\mathtt{Int}}) \quad \textbf{case}\ \mathtt{I\#}^{\gamma}\ P\ \textbf{of}\ \ \mathtt{I\#}\ x \rightarrow e = e[P/x] \qquad (\eta_{\mathtt{Int}}) \quad \textbf{case}\ e\ \textbf{of}\ \ \mathtt{I\#}\ x \rightarrow \mathtt{I\#}^{\gamma}\ x = e : \mathtt{Int}^{\gamma}$$

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

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

That is, the right-hand side of a **let** must have statically known representation and calling convention. We can add a similar derived typing rule LET-P, by composing LAM-I and LAM-P-E, for the special case when the right hand side is passive so its convention does not need to be known.

3.10 Equational theory

The equational theory for $\mathcal{IL}$ is defined in Fig. 4. It gives us a framework to reason about equality in $\mathcal{IL}$, and ultimately about the correctness of compiling $\mathcal{IL}$ to a low-level language (Theorem 1) as well as for compiling a high-level language to $\mathcal{IL}$ (Theorem 4). The rules for `Clos`/`App` (namely $\beta_{\{\}}$ and $\eta_{\{\}}$) and **case**, `I#` ($\beta_{\mathtt{Int}}$ 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 3.3) allows unrestricted η in either direction for any expression of a function type.

That leaves the reduction rule $\beta_{\rightsquigarrow}$. As is usual in a call-by-value λ-calculus, only some expressions—called *passable* or *substitutable* expressions, and here denoted by the metavariable S—can be passed to a function and substituted for its formal parameter by the $\beta_{\rightsquigarrow}$ rule. Only if the argument of a β-redex is passable does the $\beta_{\rightsquigarrow}$ rule fire. Unlike most systems, which use a syntactic definition of the expressions that can be passed or substituted, $\mathcal{IL}$ instead identifies these expressions by their type and kind, as defined by the rules for the $\Gamma \vdash e\ \textbf{pass}$ judgment. Using the type and kind of the argument to define passability, rather than only its syntax, allows us to integrate several different evaluation orders (call-by-value, call-by-name, *etc.*) within the same language.

Let us examine closely the $\Gamma \vdash e\ \textbf{pass}$ judgment. Firstly, we designate all *passive* expressions P to be *passable*. These expression forms include variables (which would always be substituted for passable expressions), constants, and applications of `I#` and `Clos`. Notice that, thus far, these are all considered values in the call-by-value λ-calculus. The grammar for P also looks through abstractions $\lambda\chi.P$ and applications $P\ \phi$, which are erased anyway during compilation, and thus have no impact on the runtime behavior of a program.

Notice how the notion of passability depends specifically on the calling convention for the type of that expression. For example, in a call-by-name setting, every expression can be substituted for a variable. So in $\mathcal{IL}$, *all* arguments that are to be lazily evaluated—which have the convention $\mathtt{Eval}^{\mathsf{L}}$—are considered passable, and hence allow $\beta_{\rightsquigarrow}$ to fire. In contrast, only expressions that are

passive (P) are values in call-by-value languages. So we can only substitute a variable with an expression of the convention $\texttt{Eval}^{\texttt{U}}$—which should be strictly evaluated—when it is passive. More generally, passive expressions are always substitutable in all of the evaluation strategies we are interested in here, so we can say something more: a passive P is passable for *all* calling conventions. This extra step is helpful in case we are dealing with an expression—like x or $\texttt{I\#}^{g}\ n$—which has an unknown calling convention but will inevitably be passable 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 3.5) such as

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

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

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

Here, we cannot apply the $\beta_{\rightsquigarrow}$ directly as before, because the argument is not passable: it has the type $\texttt{Int}^{\texttt{U}} : \texttt{TYPE}\ \texttt{PtrR}\ \texttt{Eval}^{\texttt{U}}$, and it is not passive. Instead we must first evaluate the argument; now it becomes passive, and $\beta_{\rightsquigarrow}$ can fire.

Of course the substitution done by $\beta_{\rightsquigarrow}$ is woefully inefficient: worse than just recursing down an expression, it irreparably *duplicates* the work of delayed computations, leading to an asymptotic slowdown in many cases. A better semantics cares about sharing work, which we show next in Section 4 when compiling to a lower-level representation. But we could still reason about complexity and sharing directly in $\mathcal{IL}$ using a call-by-need semantics, just like the λ-calculus [Ariola and Felleisen 1997]. For more details about the operational semantics of $\mathcal{IL}$, see Appendix A.[8]

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

4.1 Syntax and semantics of a machine language $\mathcal{ML}$

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 non-uniform representations and higher-arity functions, both of which must be monomorphic. The idea is to model a realistic machine architecture that supports multiple basic kinds of data representations (*e.g.*, pointers vs. integers), and where functions are passed multiple arguments but are otherwise first order (though function pointers may be passed as arguments and invoked). The syntax of this language, which we call $\mathcal{ML}$, is given in Fig. 5, along with its abstract machine[9].

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.
- Functions $(\lambda(\overline{y_\pi}).e)$ and their calls $(P(\overline{a}))$ pass many arguments at once, explicitly modeling multi-arity functions.
- The function components of an application form cannot be an arbitrary expression; it must be a passive expression, which is either a value V or a variable. This way, every application $P(\overline{a})$ can be resolved in at most two steps: lookup P if it is a variable, and then apply P if it is a λ-abstraction or a constant (like `error`).

Note that, as such, it is impossible to chain separate calls in a row. For example, $f(1)(2)$ is not a legal expression in $\mathcal{ML}$; instead, if f is an arity 2 function, it must be called as $f(1, 2)$, while if it is

[8] All appendices are included as anonymized supplementary material.

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

$$\pi \in \mathit{PrimRep} ::= \texttt{PtrR} \mid \texttt{IntR} \qquad \psi \in \mathit{Strictness} ::= \texttt{L} \mid \texttt{U} \qquad \eta \in \mathit{KnownCC} ::= \texttt{Eval}^{\psi} \mid \texttt{Call}[\overline{\pi}]$$

$$e \in \mathit{Expr} ::= P \mid P(\overline{a}) \mid \texttt{App}\ e(\overline{a}) \mid \textbf{case}\ e\ \textbf{of}\ \texttt{I\#}(x_{\texttt{IntR}}) \to e' \mid \textbf{let}\ x_{\pi}^{\psi} = e\ \textbf{in}\ e'$$

$$R \in \mathit{Reference} ::= \texttt{I\#}(a) \mid \texttt{Clos}^{n}\ P \mid \lambda(\overline{x_{\pi}}).e$$

$$V \in \mathit{Value} ::= c \mid \texttt{I\#}(a) \mid \texttt{Clos}^{n}\ P \qquad P \in \mathit{Passive} ::= R \mid x_{\pi} \qquad a \in \mathit{Arg} ::= c \mid x_{\pi}$$

$$K \in \mathit{StackCont} ::= \varepsilon \mid \texttt{App}(\overline{a}); K \mid \textbf{case}\ \texttt{I\#}(x_{\texttt{IntR}}) \to e; K \mid \textbf{let}\ x_{\pi}\ \textbf{in}\ e; K \mid \textbf{set}\ x; K$$

$$H \in \mathit{Store} ::= \varepsilon \mid [x := R]H \mid [x := \textbf{memo}\ e]H$$

$$m \in \mathit{MachineState} ::= \langle e \mid K \mid H \rangle \mid \texttt{error}(n) \qquad A \in \mathit{Answer} ::= \langle V \mid \varepsilon \mid H \rangle \mid \texttt{error}(n)$$

$$(\mathit{PshApp}) \quad \langle \texttt{App}\ e(\overline{a}) \mid K \mid H \rangle \mapsto \langle e \mid \texttt{App}(\overline{a}); K \mid H \rangle$$

$$(\mathit{PshCase}) \quad \langle \textbf{case}\ e\ \textbf{of}\ \texttt{I\#}(x_{\texttt{IntR}}) \to e' \mid K \mid H \rangle \mapsto \langle e \mid \textbf{case}\ \texttt{I\#}(x_{\texttt{IntR}}) \to e'; K \mid H \rangle$$

$$(\mathit{PshLet}) \quad \langle \textbf{let}\ x_{\pi}^{\texttt{U}} = e\ \textbf{in}\ e' \mid K \mid H \rangle \mapsto \langle e \mid \textbf{let}\ x_{\pi}\ \textbf{in}\ e'; K \mid H \rangle$$

$$(\mathit{LAlloc}) \quad \langle \textbf{let}\ x_{\texttt{PtrR}}^{\texttt{L}} = e'\ \textbf{in}\ e \mid K \mid H \rangle \mapsto \langle e[y_{\texttt{PtrR}}/x_{\texttt{PtrR}}] \mid K \mid [y := \textbf{memo}\ e']H \rangle$$

$$(\mathit{Call}) \quad \langle (\lambda(\overline{x_{\pi}}).e)(\overline{a}) \mid K \mid H \rangle \mapsto \langle e[\overline{a/x_{\pi}}] \mid K \mid H \rangle$$

$$(\mathit{Apply}) \quad \langle \texttt{Clos}^{n}\ P \mid \texttt{App}(\overline{a}); K \mid H \rangle \mapsto \langle P(\overline{a}) \mid K \mid H \rangle \qquad (\text{if } n = |\overline{a}|)$$

$$(\mathit{Unbox}) \quad \langle \texttt{I\#}(a) \mid \textbf{case}\ \texttt{I\#}(x_{\texttt{IntR}}) \to e; K \mid H \rangle \mapsto \langle e[a/x_{\texttt{IntR}}] \mid K \mid H \rangle$$

$$(\mathit{Move}) \quad \langle c \mid \textbf{let}\ x_{\pi}\ \textbf{in}\ e; K \mid H \rangle \mapsto \langle e[c/x_{\pi}] \mid K \mid H \rangle$$

$$(\mathit{SAlloc}) \quad \langle R \mid \textbf{let}\ x_{\texttt{PtrR}}\ \textbf{in}\ e; K \mid H \rangle \mapsto \langle e[y_{\texttt{PtrR}}/x_{\texttt{PtrR}}] \mid K \mid [y := R]H \rangle$$

$$(\mathit{Fun}) \quad \langle y_{\texttt{PtrR}}(\overline{a}) \mid K \mid [y := R]H \rangle \mapsto \langle R(\overline{a}) \mid K \mid [y := R]H \rangle$$

$$(\mathit{Look}) \quad \langle y_{\texttt{PtrR}} \mid K \mid [y := R]H \rangle \mapsto \langle R \mid K \mid [y := R]H \rangle$$

$$(\mathit{Force}) \quad \langle y_{\texttt{PtrR}} \mid K \mid [y := \textbf{memo}\ e]H \rangle \mapsto \langle e \mid \textbf{set}\ y; K \mid [y := \textbf{memo}\ e]H \rangle$$

$$(\mathit{Memo}) \quad \langle R \mid \textbf{set}\ y; K \mid [y := \textbf{memo}\ e]H \rangle \mapsto \langle R \mid K \mid [y := R]H \rangle$$

$$(\mathit{Error}) \quad \langle \texttt{error}(n) \mid K \mid H \rangle \mapsto \texttt{error}(n)$$

Fig. 5. Syntax and semantics for $\mathcal{ML}$

an arity-1 function returning a closure of an arity-1 function it must be called as (App $f(1)$)(2). The number of arguments passed at once is explicitly fixed in each λ-abstraction defining a function and call site which uses it. In other words, $\mathcal{ML}$ does not support polymorphism of function arity.

The syntax of $\mathcal{ML}$ includes annotations that make explicit the semantically important aspect of types, which are implicit in $\mathcal{IL}$, so that it is clear from the syntax how to execute programs. In particular, each variable is annotated with its representation π, which must be either a pointer (PtrR) or integer (IntR). In other words, all variables are permanently assigned a representation—intuitively, specifying how they are stored in either an integer register or a pointer on the heap. By design, $\mathcal{ML}$ does not support polymorphism over these representations because the machine's pointer registers and floating-point registers are distinct.

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 these levities.

4.2 The semantics of $\mathcal{ML}$

Executing an $\mathcal{ML}$ program involves a machine configuration of the form $\langle e \mid K \mid H \rangle$, 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 which 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 $\mathtt{App}(\overline{a}); K$. 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*, *VAlloc*), but note that we include cases for *both* lazy bindings (*LAlloc*) and strict ones (*PshLet*, *VAlloc*).

Next we have the rules for performing interesting reductions. *Apply* resolves the application of a closure by extracting the function it contains, and *Call* calls a known 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 5). 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 called function to map to a value. For other pointers, we have to check whether or not it is evaluated, and merely *Look* up values or else *Force* thunks.[10] When a forced thunk returns a value, it is *Memo*ized to share the result on future uses of that pointer.

4.3 Compilation

The compilation transformation from $\mathcal{IL}$ to the low-level machine language $\mathcal{ML}$ is given in Fig. 6. The top-level compilation function is $\mathcal{E}_\nu[\![e]\!]^\Gamma_\theta$, which compiles a typed expression $\Gamma \vdash e : \tau$ where 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 $[a_1/x_1] \ldots [a_n/x_n]$.

A key part in understanding the compilation function 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 calling convention ν of e. If it is $\mathtt{Eval}^\gamma$, 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 function match, we always fully η-expand function expressions when they are defined: either on the right-hand side of **let**s or in the body of `Closures`. Because of this dependency, this step of compilation is only defined when the calling convention is statically known (*e.g.*, it is not a variable v or partially-defined like $\mathtt{Call}[r_1, r_2, \mathit{arity}(v)]$). In any case, we next move to the main work-horse of compilation, $C[\![e]\!]^\Gamma_\theta(\overline{a})$, 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.

Even compiling constants and variables, at the top of the figure, brings in some complications. Literal constants are simply passed through, but when compiling a call to `error`, we see that

[10] 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. This means that 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:

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

Constants:

$$C[\![n]\!]^{\Gamma}_{\theta}(\varepsilon) \triangleq n \qquad C[\![\mathtt{error}]\!]^{\Gamma}_{\theta}(a) \triangleq \mathtt{error}(a)$$

Variables:

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

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

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

$$C[\![\mathtt{App}\ e]\!]^{\Gamma}_{\theta}(\overline{a}) \triangleq \mathtt{App}\ (C[\![e]\!]^{\Gamma}_{\theta}(\varepsilon))(\overline{a})$$

$$C[\![e\ \phi]\!]^{\Gamma}_{\theta}(\overline{a}) \triangleq C[\![e]\!]^{\Gamma}_{\theta}(\overline{a})$$

$$C[\![e\ P]\!]^{\Gamma}_{\theta}(\overline{a}) \triangleq C[\![e]\!]^{\Gamma}_{\theta}(\mathcal{P}[\![P]\!]^{\Gamma}_{\theta}, \overline{a}) \qquad (\text{if } \mathcal{P}[\![P]\!]^{\Gamma}_{\theta} = x_{\pi} \text{ or } \mathcal{P}[\![P]\!]^{\Gamma}_{\theta} = c)$$

$$C[\![e\ P]\!]^{\Gamma}_{\theta}(\overline{a}) \triangleq \mathbf{let}\ x^{\mathsf{U}}_{\mathtt{PtrR}} = \mathcal{P}[\![P]\!]^{\Gamma}_{\theta}\ \mathbf{in}\ C[\![e]\!]^{\Gamma}_{\theta}(x_{\mathtt{PtrR}}, \overline{a})$$

$$C[\![e\ e']\!]^{\Gamma}_{\theta}(\overline{a}) \triangleq \mathbf{let}\ x^{lev(\eta)}_{\pi} = \mathcal{E}_{\eta}[\![e']\!]^{\Gamma}_{\theta}\ \mathbf{in}\ C[\![e]\!]^{\Gamma}_{\theta}(x_{\pi}, \overline{a}) \quad (\text{if } \Gamma \vdash e' : \tau, \Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi, \text{ and } \Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta)$$

Boxing and unboxing:

$$C[\![\mathtt{I\#}^{\gamma}\ P]\!]^{\Gamma}_{\theta}(\varepsilon) \triangleq \mathcal{P}[\![\mathtt{I\#}^{\gamma}\ P]\!]^{\Gamma}_{\theta}$$

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

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

Abstractions:

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

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

$$C[\![\mathtt{Clos}^{\gamma}\ e]\!]^{\Gamma}_{\theta}(\varepsilon) \triangleq \mathcal{P}[\![\mathtt{Clos}^{\gamma}\ e]\!]^{\Gamma}_{\theta}$$

Passive expressions:

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

$$\mathcal{P}[\![P\ \phi]\!]^{\Gamma}_{\theta} \triangleq \mathcal{P}[\![P]\!]^{\Gamma}_{\theta} \quad \mathcal{P}[\![\lambda\chi.P]\!]^{\Gamma}_{\theta} \triangleq \mathcal{P}[\![P]\!]^{\Gamma}_{\theta}$$

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

Calculating known representations and conventions: $\boxed{\Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi}$ and $\boxed{\Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta}$

$$\frac{\Gamma \vdash \tau : \mathtt{TYPE}\ \rho\ \nu \quad \rho = \pi}{\Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi} \qquad \frac{\Gamma \vdash \tau : \mathtt{TYPE}\ \rho\ \nu \quad \nu = \eta}{\Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta}$$

The levity of a known convention:

$$lev(\mathtt{Eval}^{\psi}) = \psi \qquad lev(\mathtt{Call}[\overline{\pi}]) = \mathsf{U}$$

Fig. 6. Compiling $\mathcal{IL}$ to $\mathcal{ML}$

precisely one argument a is required. 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. However, note that there is different $\mathcal{ML}$ code for *evaluating* x versus *calling* it, even when there are no arguments. In other words, a function variable x with the empty calling convention `Call`[] compiles as $C[\![x]\!]^{\Gamma}_{[y_{\texttt{PtrR}}/x]\theta}(\varepsilon) = y_{\texttt{PtrR}}()$, which is a nullary function call, but a boxed integer variable x with the evaluation convention $\texttt{Eval}^{\gamma}$ compiles as $C[\![x]\!]^{\Gamma}_{[y_{\texttt{PtrR}}/x]\theta}(\varepsilon) = y_{\texttt{PtrR}}$ which merely looks up the pointer.

Closure applications are compiled straightforwardly, and an erasable arguments ϕ and binders $\lambda\chi.e$ are simply dropped. Passive expressions of $\mathcal{IL}$ can be compiled outright to passive expressions of $\mathcal{ML}$ via $\mathcal{P}[\![P]\!]^{\Gamma}_{\theta}$, which also performs type erasure. Compiling an application to a passive argument P depends on the nature of that passive argument:

- If $\mathcal{P}[\![P]\!]^{\Gamma}_{\theta}$ is a variable or constant (after erasure type), then it can be passed directly.
- Otherwise, we name the argument with a **let** (respecting the A-normal form) and pass it by reference to the function. Notice that in this case, the compiled argument will always have the form $\texttt{I\#}(a)$ or $\texttt{Clos}^{n}\ P$, which means the **let**-binding will always be represented as a pointer into the heap.[11]

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

In the case of a general application $e\ e'$ to an arbitrary argument that needs to be computed, corresponding to Lam-E, we always generate a **let** similar to the second case for $e\ P$. However, for Lam-E, we need to determine the representation, calling convention, *and* levity of the binding, which could truly be anything. This is why we require the additional restrictions in Lam-E that correspond to the highlighted side conditions of this case.

4.4 Correctness of compilation

Notice how the same polymorphism restrictions used in the typing rules also appear during compilation. Even though the defined compilation transformation is partial (that is, not every syntactically valid expression can be compiled), all well-typed $\mathcal{IL}$ expressions with a known calling convention have a defined compilation to the lower-level, levity-monomorphic $\mathcal{ML}$. That is, we want to ensure that the compilation $\mathcal{E}_{\eta}[\![e]\!]$ is well-defined for any well-typed closed expression $\vdash e : \tau : \mathsf{TYPE}\ \rho\ \eta$, where the syntax of known calling conventions η is from Fig. 5.

In fact, we allow for a little more levity polymorphism during compilation: $\mathcal{E}_{\texttt{Eval}^{g}}[\![e]\!]$, for an unknown levity g, is also allowed. That's because we generate the code that will be executed only when the expression is evaluated: in other words, when a computation is forced, there is no difference between eager or lazy. This added flexibility is essential for compiling levity polymorphic expressions appearing in strict contexts, such as in the discriminant of a **case** or argument of `App`. Although implicit, the C compilation function makes the same assumptions on the known convention and strictness of the expression it compiles. In contrast, the $\mathcal{P}$ function does not assume the expression is being evaluated, because it is only 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 multi-arity functions) without erroneously introducing extra strictness.

[11] It may be that in a more feature-rich language, this case could involve representations other than just a single pointer. However, the value returned from a passive expression is syntactically manifest. As such, the representation is always be apparent from the syntax of the expression, which means we do not require additional typing information here.

During compilation, we occasionally require the ability to calculate a known representation (π) or calling convention (η) for a sub-expression. This appears in Fig. 6 as the highlighted side conditions $\Gamma \vdash \tau \overset{rep}{\rightsquigarrow} \pi$ and $\Gamma \vdash \tau \overset{conv}{\rightsquigarrow} \eta$, respectively. In general, these calculations could fail if the representation of 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. The places where this requirement appears corresponds exactly to the highlighted monomorphism restrictions in Fig. 2. For more details on the invariants for compiling open expressions, see Appendix B.

Theorem 1 (Closed Compilation). *If* $\vdash e : \tau$ *and* $\vdash \tau : \mathtt{TYPE}\ \rho\ \nu$ *then* $\mathcal{E}_\nu[\![e]\!]$ *is defined.*

This theorem just states when compilation translates a closed $\mathcal{IL}$ expression to $\mathcal{ML}$ code. We should also expect that compilation preserves the behavior of $\mathcal{IL}$ expressions as well. In other words, if an expression is equal to some answer in $\mathcal{IL}$, then executing the compiled code should give the same answer in $\mathcal{ML}$. But note that we are not interested in evaluating functions directly—they must be called, not evaluated!—so answers will be of some `Evaluatable` types like (un)boxed integers, which are simple enough values to line up on the nose. For more details on the semantic relationship between $\mathcal{IL}$ and $\mathcal{ML}$, see Appendices A and B in the supplementary material.

Theorem 2 (Soundness and Completeness)**.**

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

5 DYNAMIC ARITY RAISING

So far, both the intermediate language $\mathcal{IL}$ and lower-level abstract machine $\mathcal{ML}$ assume that the arity of all functions are statically known at compile-time. This is not the most ideal situation, especially since it has been shown that arity mismatches of "unknown" function calls can be efficiently accommodated in practical implementations [Marlow and Peyton Jones 2004]. This section thus proposes an optional extension to $\mathcal{ML}$, granting more flexibility at a small runtime cost when unpacking closures. Such additional flexibility requires only a little dynamic checking at run-time. Applying too few arguments creates a partial application, and applying too many is broken down into two (or more) separate calls. Both of these possibilities are already accounted for by Marlow and Peyton Jones [2004] with "unknown" function calls. In terms of $\mathcal{ML}$, we can extend its grammar with partial applications of the form $\mathtt{Clos}^n\ f(\overline{a})$, where $n > 0$ is the number of remaining arguments expected before the function f can be called, and $\overline{a}$ are the arguments applied so far. We can now express the extra rules for dynamic handling a run-time arity mismatch:

$$(\mathit{Apply}) \quad \langle \mathtt{Clos}^n\ P(\overline{a}) \mid \mathtt{App}(\overline{a}'); K \mid H \rangle \mapsto \langle P(\overline{a}, \overline{a}') \mid K \mid H \rangle \qquad (\text{if } |\overline{a}'| = n)$$

$$(\mathit{PApp}) \quad \langle \mathtt{Clos}^n\ P(\overline{a}) \mid \mathtt{App}(\overline{a}'); K \mid H \rangle \mapsto \langle \mathtt{Clos}^{n-|\overline{a}'|}\ P(\overline{a}, \overline{a}') \mid K \mid H \rangle \qquad (\text{if } |\overline{a}'| < n)$$

$$(\mathit{OApp}) \quad \langle \mathtt{Clos}^n\ P(\overline{a}) \mid \mathtt{App}(\overline{a}', \overline{a}''); K \mid H \rangle \mapsto \langle P(\overline{a}, \overline{a}') \mid \mathtt{App}(\overline{a}''); K \mid H \rangle \qquad (\text{if } |\overline{a}'| = n, |\overline{a}''| > 0)$$

In practice, this lets us get away with *fusing* adjacent chains of `Closures` and `Applications`, so that the runtime system can dynamically choose a better calling convention when possible.

For example, consider the following two different closure wrappers around the `plus#` function:

$$\begin{array}{ll} \mathit{plus}_1 : {}^{\mathtt{L}}\{\mathtt{Int\#} \rightsquigarrow {}^{\mathtt{L}}\{\mathtt{Int\#} \rightsquigarrow \mathtt{Int\#}\}\} & \mathit{plus}_2 : {}^{\mathtt{L}}\{\mathtt{Int\#} \rightsquigarrow \mathtt{Int\#} \rightsquigarrow \mathtt{Int\#}\} \\ \mathit{plus}_1 = \mathtt{Clos}^{\mathtt{L}}\ \lambda x{:}\mathtt{Int\#}.\ \mathtt{Clos}^{\mathtt{L}}\ \lambda y{:}\mathtt{Int\#}.\ \mathtt{plus\#}\ x\ y & \mathit{plus}_2 = \mathtt{Clos}^{\mathtt{L}}\ \lambda x{:}\mathtt{Int\#}.\lambda y{:}\mathtt{Int\#}.\ \mathtt{plus\#}\ x\ y \end{array}$$

plus_1 is written to receive one argument at a time, whereas plus_2 expects two arguments at once, and the difference between the two calling conventions is expressed in *both* the expression and its

type. These arities are made more explicit when compiling to the lower level:

$$\mathcal{E}_{\mathtt{Eval}^{\mathtt{L}}}[\![\mathit{plus}_1]\!] = \mathtt{Clos}^1\ \lambda(x_{\mathtt{IntR}}).\ \mathtt{Clos}^1\ \lambda(y_{\mathtt{IntR}}).\ \mathtt{plus\#}(x_{\mathtt{IntR}}, y_{\mathtt{IntR}})$$
$$\mathcal{E}_{\mathtt{Eval}^{\mathtt{L}}}[\![\mathit{plus}_2]\!] = \mathtt{Clos}^2\ \lambda(x_{\mathtt{IntR}}, y_{\mathtt{IntR}}).\ \mathtt{plus\#}(x_{\mathtt{IntR}}, y_{\mathtt{IntR}})$$

If we support only the *Apply* rule at runtime, as done in Fig. 5, then this difference is important not only for performance, but *also* correctness: *Apply* fails if the arity of a function does not match the arity of an application exactly. However, with *PApp* and *OApp*, it is correct to use plus_1 and plus_2 interchangeably at runtime, albeit with different performance. Applying plus_2 to one argument as in $\mathtt{App}\ \mathit{plus}_2(1)$ dynamically creates a partial application object $\mathtt{Clos}^2\ (\lambda(x_{\mathtt{IntR}}, y_{\mathtt{IntR}}).\ \mathtt{plus\#}(x, y))(1)$, whereas applying plus_1 to two arguments hits the over-application case:

$$\langle \mathit{plus}_1 \mid \mathtt{App}(1,2); K \mid H\rangle \mapsto_{\mathit{OApp}} \langle (\lambda(x_{\mathtt{IntR}}).\ \mathtt{Clos}^1\ \lambda(y_{\mathtt{IntR}}).\ \mathtt{plus\#}(x,y))(1) \mid \mathtt{App}(2); K \mid H\rangle$$

In other words, it is safe to treat these two expressions interchangeably in our extended $\mathcal{ML}$.

But notice: if we compile only well-typed $\mathcal{IL}$ expressions, then over application (*OApp*) and under application (*PApp*) will never happen! With the support that over- and under-application give for dynamically checking arity at runtime, the type system of $\mathcal{IL}$ is overly conservative. Programs that are safe to execute in $\mathcal{ML}$ would be rejected in $\mathcal{IL}$. That's because the actual arity can be checked at runtime, so an arity 2 closure is treated like it has arity 1 (by *PApp*) and vice versa (by *OApp*).

What is missing is a way to say that plus_1 and plus_2, though they are different, are treated identically at runtime: under this extension, both would have their arity dynamically checked before any function is called. There is no harm in using values of one type in any context expecting the other. In more general terms, if we always dynamically check the arity of closures, we can safely equate the following two types without the need for wrapping and unwrapping closures:

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

One way is to capture both of these two operations as a type equality, or a type-safe coercion [Breitner et al. 2016]: the two types are represented identically at runtime as closure objects, with the only difference being the arity count stored inside—which will be checked dynamically.

Correspondingly, consider applying closures of different types to the same argument twice:

$$\mathit{dup}_1 : {}^{\mathtt{L}}\{\mathtt{Int\#} \rightsquigarrow {}^{\mathtt{L}}\{\mathtt{Int\#} \rightsquigarrow \mathtt{Int\#}\}\} \rightsquigarrow \mathtt{Int\#} \rightsquigarrow \mathtt{Int\#} \qquad \mathit{dup}_2 : {}^{\mathtt{L}}\{\mathtt{Int\#} \rightsquigarrow \mathtt{Int\#} \rightsquigarrow \mathtt{Int\#}\} \rightsquigarrow \mathtt{Int\#} \rightsquigarrow \mathtt{Int\#}$$
$$\mathit{dup}_1\ f\ x = \mathtt{App}\ (\mathtt{App}\ f\ x)\ x \qquad \mathit{dup}_2\ f\ x = \mathtt{App}\ f\ x\ x$$

These compile to the following $\mathcal{ML}$ functions:

$$\mathcal{E}_{\mathtt{Call[PtrR,Int\#]}}[\![\mathit{dup}_1]\!] = \lambda(f_{\mathtt{PtrR}}, x_{\mathtt{IntR}}).\mathtt{App}\ (\mathtt{App}\ f_{\mathtt{PtrR}}(x_{\mathtt{IntR}}))(x_{\mathtt{IntR}})$$
$$\mathcal{E}_{\mathtt{Call[PtrR,Int\#]}}[\![\mathit{dup}_2]\!] = \lambda(f_{\mathtt{PtrR}}, x_{\mathtt{IntR}}).\mathtt{App}\ f_{\mathtt{PtrR}}(x_{\mathtt{IntR}}, x_{\mathtt{IntR}})$$

By checking arities at runtime, both dup_1 and dup_2 can be safely applied to either plus_1 or plus_2. This can be necessary when the *plus* operations need to be stored in a list that requires all elements be of the same type. If it happens that plus_2 is stored with plus_1's type, and then later dup_2 is applied to it with dup_1's type, then we still get the benefit of passing two arguments at once at runtime, even if our type system cannot ensure it statically.

In the end, the details of dynamically-resolved, higher-arity, function calls can be fully captured by the type system. The arity of a function call is described *exactly* by the type and kind of the function, and the program must provide the right number of arguments and binders. However, as long as there is a top-most lifted abstraction, which corresponds to storing additional arity information at run-time, we can freely convert between functions of different arities. In other words, kinds fully capture calling conventions in programs, as the kind of a boxed closure does not mention any arity and thus must be subject to further checks.

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

$$[\![\mathtt{Int}]\!] \triangleq \mathtt{Int}^{\mathsf{L}} \quad [\![t]\!] \triangleq t \quad [\![x]\!] \triangleq x \quad [\![n]\!] \triangleq \mathtt{I\#}^{\mathsf{L}}\ n$$
$$[\![\tau \to \sigma]\!] \triangleq {}^{\mathsf{L}}\{[\![\tau]\!] \rightsquigarrow [\![\sigma]\!]\} \quad [\![\lambda x{:}\tau.e]\!] \triangleq \mathtt{Clos}^{\mathsf{L}}\ \lambda x{:}[\![\tau]\!].[\![e]\!] \quad [\![e\ e']\!] \triangleq (\mathsf{App}\ [\![e]\!])\ [\![e']\!]$$
$$[\![\forall t.\tau]\!] \triangleq \forall t{:}\star.[\![\tau]\!] \quad [\![\lambda t.e]\!] \triangleq \lambda t{:}\star.[\![e]\!] \quad [\![e\ \tau]\!] \triangleq [\![e]\!]\ [\![\tau]\!]$$

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

$$[\![\mathtt{Int}]\!] \triangleq \mathtt{Int}^{\mathsf{U}} \quad [\![t]\!] \triangleq t \quad [\![x]\!] \triangleq x \quad [\![n]\!] \triangleq \mathtt{I\#}^{\mathsf{U}}\ n$$
$$[\![\tau \to \sigma]\!] \triangleq {}^{\mathsf{U}}\{[\![\tau]\!] \rightsquigarrow [\![\sigma]\!]\} \quad [\![\lambda x{:}\tau.e]\!] \triangleq \mathtt{Clos}^{\mathsf{U}}\ \lambda x{:}[\![\tau]\!].[\![e]\!] \quad [\![e\ e']\!] \triangleq (\mathsf{App}\ [\![e]\!])\ [\![e']\!]$$
$$[\![\forall t.\tau]\!] \triangleq {}^{\mathsf{U}}\{\forall t{:}\star.[\![\tau]\!]\} \quad [\![\lambda t.e]\!] \triangleq \mathtt{Clos}^{\mathsf{U}}\ \lambda t{:}\star.[\![e]\!] \quad [\![e\ \tau]\!] \triangleq (\mathsf{App}\ [\![e]\!])[\![\tau]\!]$$

Fig. 7. Compiling call-by-name and call-by-value System F to $\mathcal{IL}$

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. That is indeed possible, by simply having two different closure types (with their introduction and elimination forms). Then, an optimizing compiler, or an expert user, can select the appropriate form for the best performance.

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

$\mathcal{IL}$ is, by design, a fairly low-level language that makes fine distinctions about representation, levity and so on. This allows it to act as a target for *both* eager *and* lazy languages. To make this claim concrete, we now give translations for call-by-name System F (Section 6.1) and call-by-value System F (Section 6.2) into $\mathcal{IL}$.

6.1 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 is capable of capturing all the types of the source language, namely $\star$ = TYPE PtrR Eval$^{\mathsf{L}}$. Each source-language type τ will be represented by an $\mathcal{IL}$ type $[\![\tau]\!]$ with kind $\star$; that is, a pointer to a lifted value, perhaps a thunk.

Figure 7 gives this type translation. To get the correct call-by-name semantics for numbers, we have to use the boxed integer type Int$^{\mathsf{L}}$, which happily has the correct kind. However, even though the function type $[\![\tau]\!] \rightsquigarrow [\![\sigma]\!]$ has the correct semantics, it has the wrong kind TYPE PtrR Call[PtrR]. Therefore, the translation coerces the calling convention with a closure type. The only change to polymorphic type abstraction is to say that the type variable t ranges over the "uniform" kind $\star$.

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 more descriptive kinds.

Call-by-need evaluation is often considered to be the more "practical" version of call-by-name, because it has better asymptotic complexity in many cases but still gives the same result for purely functional programs [Ariola and Felleisen 1997]. We make the same conflation here, where we interpret Eval$^{\mathsf{L}}$ to mean call-by-name in the high-level equational theory (Fig. 4), but actually

Call-by-name substitutable values: $V ::= e$

Call-by-value substitutable values: $V ::= x \mid \lambda x{:}\tau.e \mid \lambda t.e$

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

$$(\beta_{\to}) \quad (\lambda x{:}\tau.e)\ V = e[V/x] \qquad (\eta_{\to}) \quad \lambda x{:}\tau.(V\ x) = V : \tau \to \sigma$$

$$(\beta_{\forall}) \quad (\lambda t.V)\ \tau = e[\tau/t] \qquad (\eta_{\forall}) \quad \lambda t.(V\ t) = V : \forall t.\sigma$$

$$(name) \quad (\lambda x{:}\tau.e\ x)\ e' = e\ e'$$

Fig. 8. Equational theory of System F; call-by-value and call-by-name

implement call-by-need evaluation for $\texttt{Eval}^{\texttt{L}}$ in the machine (Fig. 5). So to compile a pure call-by-need programming language, like Haskell, one can just apply the call-by-name compilation scheme directly, and get the correct performance characteristics.

A realistic compiler for a call-by-need source language should also interpret the lifted levity L as call-by-need in $\mathcal{IL}$ itself. For example, the call-by-name $(\beta_{\rightsquigarrow})$ rule in Figure 4 duplicates the argument S, which might duplicate an arbitrary amount of work. It is well known how to adjust the equational theory to enshrine call-by-need [Ariola and Felleisen 1997], and we do not elaborate here.

6.2 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 $\star = \texttt{TYPE PtrR Eval}^{\texttt{U}}$ for call-by-value evaluation. As before, we can compile source-level types following the invariant that $[\![\tau]\!] : \star$, as again shown in Fig. 7.

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 wrapped in a closure, as in the call-by-name case, but this time unlifted.

The translation of polymorphism is more complex, however, due to the standard semantics of the call-by-value System F. With call-by-value evaluation, the abstraction $\lambda t.\bot$ is a value, despite that $\bot$ diverges, whereas in call-by-name they would be η-equivalent. We need to make sure 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 in expressions is 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 $\texttt{Clos}^{\texttt{U}}\ (\lambda t{:}\star.[\![e]\!])$, which is essential when side effects or non-termination enters the picture. For example, evaluating $\lambda t.\texttt{error}\ t\ n$ returns immediately in call-by-value System F, but the corresponding $\lambda t{:}\star.\texttt{error PtrR U}\ t$ 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 $\lambda().e$.

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

Compiling call-by-name and call-by-value System F into $\mathcal{IL}$ are both correct, in terms of the preservation of types and the preservation of equalities between expressions from the source to the intermediate language. We assume the standard type system for System F, and review the calculus'

call-by-name and -value axiomatic semantics in Fig. 8. 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 $\beta_{\rightarrow}$ in the call-by-name semantics. The preservation of types is straightforward, where the compilation of a typing environment $[\![\Gamma]\!]$ is defined pointwise.

Theorem 3 (Type Preservation). *If* $\Gamma \vdash e : \tau$ *is derivable then so is* $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]$.

More interesting is the translation of equalities from the source to the intermediate language, for which we use the axiomatic semantics given in Fig. 4. For more details of the proof, see Appendix C.

Theorem 4 (Soundness and Completeness). $\Gamma \vdash e = e' : \tau$ *in call-by-name System F if and only if* $[\![\Gamma]\!] \vdash [\![e]\!] = [\![e']\!] : [\![\tau]\!]$ *via the call-by-name compilation, and likewise for call-by-value.*

7 USER-DEFINED LEVITY-POLYMORPHIC DATA TYPES

Thus far we have had a single built-in boxed data type, `Int`. It is easy to see how to generalize the idea to arbitrary user-defined algebraic data types [Graf 2020]. For example, rather than having `Int` built-in, we might define it like this

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

Now, the `Int` type is parameterized by a chosen levity, 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 strictly or lazily evaluated, and whose cells contain elements of any arity or evaluation strategy:

$$\begin{array}{l}\textbf{data}\ \texttt{List}\ (g{:}\texttt{Lev})\ (v{:}\texttt{Conv})\ (a{:}\texttt{TYPE}\ \texttt{PtrR}\ v) : \texttt{TYPE}\ \texttt{PtrR}\ (\texttt{Eval}\ g)\ \textbf{where}\\ \quad \texttt{Nil} : \texttt{List}\ g\ v\ a\\ \quad \texttt{Cons} : a \rightsquigarrow \texttt{List}\ g\ v\ a \rightsquigarrow \texttt{List}\ g\ v\ a\end{array}$$

Despite the 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 of any levity:

$$\begin{array}{lll} \mathit{sum} & : & \forall(g{:}\texttt{Lev}).\texttt{List}\ g\ (\texttt{Eval}\ g)\ (\texttt{Int}\ g) \rightsquigarrow \texttt{Int}\ g\\ \mathit{sum}\ \texttt{Nil} & = & \texttt{I\#}\ 0\\ \mathit{sum}\ (\texttt{Cons}\ (\texttt{I\#}\ x)\ xs) & = & \textbf{case}\ \mathit{sum}\ xs\ \textbf{of}\ \texttt{I\#}\ y \longrightarrow \texttt{I\#}\ (\texttt{plus\#}\ x\ y)\end{array}$$

This polymorphic definition is possible because the *sum* function is completely strict, no matter if it is given an evaluated list or an unevaluated thunk, the entire thing will be added together before a value is returned. Therefore, the same code can be used for either strict or lazy evaluation. Other functions, such as mapping over a list, could not be levity polymorphic in the same way, because the code is very different where a lazy map will delay the recursive call and first return a `Cons` cell, whereas a strict map will first recurse before returning anything.

Because this `List` type is also polymorphic in calling convention, it can be used to contain lists of functions of statically-known arity, as foreshadowed in Section 3.4. For example:

$$\texttt{Cons plus\# Nil} : \texttt{List}\ (\texttt{Call}[\texttt{IntR}, \texttt{IntR}])\ (\texttt{Int\#} \rightsquigarrow \texttt{Int\#} \rightsquigarrow \texttt{Int\#})$$

8 RELATED WORK

The system presented in this paper is the culmination and consolidation of several independent lines of work on expressing performance issues directly in an intermediate language. The underlying

theme of each of these topics 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 lifted from unlifted types goes back to [Peyton Jones and Launchbury 1991], and has been used to great effect in GHC for nearly three decades. For most of that time the distinction has been static, but recent work has 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* — indeed, it recognizes no such distinction. Our contribution here is to separate the two completely, and show that one might want to be polymorphic in one but not the other.

One of the main requirements we have followed is to generate only *one* piece of code for every polymorphic definition. This requirement means that there are certain definitions that must be rejected, because the compilation depends on a choice of levity. An alternative approach by [Dunfield 2015] accepts more uses of levity polymorphism, but at the cost of generating *different* code for each choice—an exponential blowup of code size in practice—avoided by our approach.

8.2 Optimizing curried functions

Previous work has established methods for optimizing curried function calls dynamically at runtime, in order to avoid the overhead of naïvely calling $((f\ 1)\ 2)\ 3$ by passing one argument at a time, taking extra steps and allocating spurious closures in between each application. In practice, f will often expect all three arguments before doing any interesting work, so those calls should be fused when possible. This 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 have been able capture this dynamic type of optimization within the syntax and types of programs, as described in Section 5.

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 do those optimizations statically at compile time. Of course, this is easy to do when the compiler can statically find the definition of the called function [Marlow and Peyton Jones 2004]. However, 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 3.4), and polymorphism over calling conventions (Section 3.6). Another difference is that Downen et al. [2019] had two function arrows, $(\tau \to \sigma)$ 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: $\tau \to \sigma \approx \{\tau \rightsquigarrow \sigma\}$ and $\{\tau\} \approx () \to \tau$. The approach here has a greater economy of concepts, and a nice correspondence with `Int#` and $\texttt{Int}^{\gamma}$. 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 levities. Indeed GHC goes further: rather than having a stratified zoo of different sorts of things (types,

kinds, levities, representations, calling conventions) as we do in Fig. 1, they are all *types* [Weirich et al. 2013] kept separate by their *kinds*. This is a fantastic simplification, and immediately allows polymorphism over all these conceptually-different things. This does mean, however, that in GHC it is hard *not* to have polymorphism! Returning to Section 3.4, it would be hard to prevent a forall-quantified type variable from being instantiated by an arrow type, because you would have to say something like "this quantified variable can have any kind *other than* `Call`". So GHC's infrastructure strongly encourages fully-fledged polymorphism, just as we present in this paper.

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.

One of the connections that previous work on unboxed types and extensional functions has noticed is that *lifting*—in the sense of denotational semantics—corresponds to a *mismatch* between machine primitives and the semantics of a programming language. The unlifted versions of types, like integers, can be implemented more directly—and therefore more efficiently—in a machine. But preventing lifting is different depending on the type: unlifted integers need to be call-by-value whereas unlifted function chains need to be call-by-name. The first way to reconcile this tension 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 that foundational line of work.

The same correspondence between types and evaluation also corresponds to *polarity* developed for focused proof search [Andreoli 1992; Laurent 2002]. Polarity and focusing has been found to correspond to functional programming [Zeilberger 2008, 2009], as well as semantics and computation [Munch-Maccagnoni 2009, 2013]. More recently, these mixed evaluation strategy languages have been extended with more 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" ($\texttt{Int} = \Uparrow\texttt{Int\#}$) and the unknown function type constructor to a "down shift" ($\{\tau \rightsquigarrow \sigma\} = \Downarrow(\tau \rightsquigarrow \sigma)$). Note that for the sake of usability, $\mathcal{IL}$ performs 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 $\uparrow \texttt{Int}$), expressed by the encoding $\{\texttt{Int\#}\} = \Downarrow\uparrow\texttt{Int\#}$.

9 CONCLUSION

This paper illustrates a cohesive system for including low-level details—specifically *representation*, *levity*, *calling convention*, 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 directly capture performance in the intermediate language, we would also like to be able to characterize the cost of computation in its semantics as well.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

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

Arvind and Kattamuri Ekanadham. 1988. Future Scientific Programming on Parallel Machines. *J. Parallel and Distrib. Comput.* 5, 5 (1988), 460–493.

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, 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.

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

$$
\begin{aligned}
A \in \mathit{Answer} &::= V \mid E[\mathtt{error}\ \rho\ \gamma\ \tau\ V] \\
V \in \mathit{Value} &::= x\ \overline{\phi} \mid c\ \overline{\phi} \mid \mathtt{I\#}^{\gamma}\ V \mid \mathtt{Clos}^{\gamma}\ e \mid \lambda x{:}\tau.e \mid \lambda\chi.V \\
E \in \mathit{EvalCxt} &::= \Box \mid e\ E \mid E\ e \mid \mathtt{App}\ E \mid \mathtt{I\#}^{\gamma}\ E \mid \mathbf{case}\ E\ \mathbf{of}\ \mathtt{I\#}\ x \to e \mid E\ \phi \mid \lambda\chi.E
\end{aligned}
$$

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

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

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

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

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

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

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

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):

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

Additionally, the operational steps are closed under evaluation contexts:

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

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 $\Gamma, v \vdash e : \tau$ *and* $\Gamma \vdash \nu$ **conv** *then* $\Gamma[\nu/v] \vdash e[\nu/v] : \tau[\nu/v]$.

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

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

(1) $e \in$ *Answer, 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

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

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

$$\begin{aligned}
V \in \mathit{Value} &::= c\ \overline{\phi} \mid R \mid \lambda\chi.V \\
R \in \mathit{Reference} &::= \mathtt{I\#}^{\gamma}\ a \mid \mathtt{Clos}^{\gamma}\ e \mid \lambda x{:}\tau.e \mid \lambda\chi.R \\
a \in \mathit{Arg} &::= c\ \overline{\phi} \mid x\ \overline{\phi} \mid \lambda\chi.a \\
A \in \mathit{Answer} &::= H[V] \mid E[\mathtt{error}\ \rho\ \gamma\ \tau\ a] \\
E \in \mathit{EvalCxt} &::= \Box \mid F[E] \mid B[E] \\
H \in \mathit{ClosCxt} &::= \Box \mid B[H] \\
F \in \mathit{FrameCxt} &::= \Box\ a \mid \mathtt{App}\ \Box \mid \mathbf{case}\ \Box\ \mathbf{of}\ \mathtt{I\#}\ x \to e \mid \mathbf{let}\, x{:}\tau = \Box\, \mathbf{in}\, e \mid \Box\ \phi \\
B \in \mathit{BindCxt} &::= \mathbf{let}\, x{:}\tau = e\, \mathbf{in}\, \Box \mid \lambda\chi.\Box
\end{aligned}$$

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.

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

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

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:

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

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

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

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

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

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

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.

$$(\beta_{\rightsquigarrow}) \quad \Gamma \vdash (\lambda x{:}\tau.e)\ a \mapsto e[a/x] : \sigma$$

$$(\beta_{\forall}) \quad \Gamma \vdash (\lambda \chi.A)\ \phi \mapsto A[\phi/\chi] : \sigma[\phi/\chi]$$

$$(\beta_{\{\}}) \quad \Gamma \vdash \mathtt{App}\ (\mathtt{Clos}^{\gamma}\ e) \mapsto e : \sigma$$

$$(\beta_{\mathtt{Int}}) \quad \Gamma \vdash \mathbf{case}\ \mathtt{I\#}^{\gamma}\ a\ \mathbf{of}\ \mathtt{I\#}\ x \to e \mapsto e[a/x] : \sigma$$

$$(\mathit{rename}) \quad \Gamma \vdash \mathbf{let}\ x{:}\tau = a\ \mathbf{in}\ e \mapsto e[a/x] : \sigma$$

$$(\mathit{name}_{\mathtt{I\#}}) \quad \Gamma \vdash \mathtt{I\#}^{\gamma}\ e \mapsto \mathbf{let}\ x{:}\mathtt{Int\#} = e\ \mathbf{in}\ \mathtt{I\#}^{\gamma}\ x : \mathtt{Int}^{\gamma} \qquad (\text{if } e \notin \mathit{Arg})$$

$$(\mathit{name}) \quad \Gamma \vdash e'\ e \mapsto \mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ e'\ x : \sigma \qquad (\text{if } \Gamma \vdash e : \tau \text{ and } e \notin \mathit{Arg})$$

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 an answer.

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

$$\frac{\Gamma \vdash F \mathbin{@} \mathbf{let}\ x{:}\tau' = e\ \mathbf{in}\ A : \tau \implies \sigma \quad \Gamma \vdash \mathbf{let}\ x{:}\tau' = e\ \mathbf{in}\ A\ \mathbf{ans}}{\Gamma \vdash F[\mathbf{let}\ x{:}\tau' = e\ \mathbf{in}\ A] \mapsto \mathbf{let}\ x{:}\tau' = e\ \mathbf{in}\ F[A] : \sigma}\ \mathit{comm}$$

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

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) $e \in \mathit{Answer}$, *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) $e \in$ *Answer, 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:

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

$$(unname) \qquad \Gamma \vdash (\lambda x{:}\tau.e'\ x)\ e \mapsto e'\ e : \sigma \qquad (\text{if } x \notin \mathrm{FV}(e'))$$

$$(unname_{\mathtt{I\#}}) \quad \Gamma \vdash (\lambda x{:}\mathtt{Int\#}.\mathtt{I\#}^{\gamma}\ x)\ e \mapsto \mathtt{I\#}^{\gamma}\ e : \mathtt{Int}^{\gamma}$$

Lemma 11 (Context Unwinding). *(1) If* $\Gamma \vdash E_e \mathbin{@} e_e : \tau \overset{\Gamma', \Gamma''}{\Longrightarrow} \sigma$ *in the call-by-need* $\mathcal{IL}$, $\Gamma \vdash_{\mathcal{U}} E_e[e_e] \sim e'_s : \sigma$, *and* $\Gamma \nvdash e_e \ \mathbf{val}$, *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* $\Gamma \vdash E_s \mathbin{@} e_s[\theta] : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ *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 $\Gamma \vdash H_e \mathbin{@} e_e : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ *and* $\Gamma \vdash e_e \ \mathbf{val}$ *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.
- $\Gamma \vdash \mathbf{let}\, x{:}\tau' = e'_e \,\mathbf{in}\, E_e \mathbin{@} e_e : \tau \overset{\Gamma', \Gamma'', x:\tau'}{\Longrightarrow} \sigma$ because $\Gamma, x : \tau' \vdash E_e \mathbin{@} e_e : \tau \overset{\Gamma', \Gamma''}{\Longrightarrow} \sigma$ and $\Gamma \vdash e'_e \ \mathbf{bind}$. 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

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

In other words, we reduce to

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

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 $\Gamma \vdash \tau' : \mathsf{TYPE}\ \rho\ \mathsf{Eval}^{\mathsf{U}}$. 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 $\Gamma \vdash \tau' : \mathsf{TYPE}\ \rho\ \mathsf{Eval}[\mathsf{L}]$. 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

$$\Gamma, x : \tau' \vdash e_s \mapsto^* E_{s1}[e_{s1}[\theta]] : \sigma \qquad \Gamma \vdash e'_s \mapsto^* E_{s2}[e''_s[\theta]] : \tau'$$

$$\Gamma, x : \tau' \vdash E_{s1} \mathbin{@} e_{s1}[\theta] : \tau' \overset{\Gamma_1}{\Longrightarrow} \sigma \qquad \Gamma \vdash E_{s2} \mathbin{@} e''_s[\theta] : \tau \overset{\Gamma_2}{\Longrightarrow} \tau'$$

$$\Gamma, x : \tau' \vdash_{\mathcal{U}} x \sim e_{s1} : \tau' \qquad \Gamma, \Gamma_2 \vdash_{\mathcal{U}} e_e \sim e''_s : \tau$$

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

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

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

$$\Gamma, x : \tau', \overline{y_i : \tau'} \vdash e_s \mapsto^* E_{s1}[e_{s1}[\theta]] : \sigma \qquad \Gamma \vdash e'_s \mapsto^* E_{s2}[e''_s[\theta]] : \tau'$$

$$\Gamma, x : \tau', \overline{y_i : \tau'} \vdash E_{s1} \mathbin{@} e_{s1}[\theta] : \tau' \overset{\Gamma_1}{\Longrightarrow} \sigma \qquad \Gamma \vdash E_{s2} \mathbin{@} e''_s[\theta] : \tau \overset{\Gamma_2}{\Longrightarrow} \tau'$$

$$\Gamma, x : \tau', \overline{y_i : \tau'} \vdash_{\mathcal{U}} x \sim e_{s1} : \tau' \qquad \Gamma, \Gamma_2 \vdash_{\mathcal{U}} e_e \sim e''_s : \tau$$

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 (Answer Preservation). *If* $\Gamma \vdash_{\mathcal{U}} A \sim e : \tau$ *then* $e \in$ *Answer, and if* $\Gamma \vdash_{\mathcal{U}} e \sim A : \tau$ *then* $e \in$ *Answer. Likewise, if* $\Gamma \vdash_{\mathcal{N}} A \sim e : \tau$ *then* $\Gamma \vdash e \mapsto^* A' : \tau$*, and if* $\Gamma \vdash_{\mathcal{N}} e \sim A : \tau$ *then* $e \in$ *Answer.*

Proof. The first part about $\mathcal{U}$ follows immediately from Lemma 11 and the fact that answers 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 $\Gamma \vdash \tau : \mathtt{TYPE}\; \rho\; \mathtt{Eval}^{\mathsf{U}}$ (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 $\theta' = [e'_{s1}/y_1, \ldots]$):
  - 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 $\Gamma \vdash_{\mathcal{U}} e'_e \sim e'_s : \tau'$.

□

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 an answer in the case of $\beta_\forall$. Note that it is possible that a redex of the form $F[e]$ (where e **ans**) might have a **let** inserted in between the frame context and the answer 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

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

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

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

  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 $\theta' = [e'_{s1}/y_1, \ldots]$):
  - 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 $\Gamma, \overline{y_i : \tau'} \vdash E_{s1} \mathbin{@} e_{s1} : \tau \overset{\Gamma'}{\Longrightarrow} \sigma$ 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 3.10) 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 answer types τ_{ans}:

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

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 standarizaton. First, let reduction relation

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

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 an answer. 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 $\texttt{Eval}^{\texttt{U}}$ rather than *passive expressions*. That's because all passive expressions evaluate to values, anyway.

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

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

Proof. Follows from Theorem 6. □

Definition 2 (Internal Reduction). An *internal reduction*, written $\Gamma \vdash e \rightharpoonup e' : \tau$, is any reduction $\Gamma \vdash e \longrightarrow e' : \tau$ 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 answers.

Lemma 17 (Answer Stability).

(1) If $\Gamma \vdash e \rightharpoonup e' : \tau$, *then* $e \in$ *Answer if* e' *is, and* $\Gamma \vdash e$ **val** *if* e' *is.*
(2) If $\Gamma \vdash e \longrightarrow e' : \tau_{ans}$, *then* $e' \in$ *Answer if* e *is, and* $\Gamma \vdash e'$ **val** *if* e *is.*

Proof. By induction on the syntax of answers 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-answer expression. □

Definition 3 (Parallel Reduction). The *parallel reduction relation*, written $\Gamma \vdash e \Longmapsto e' : \tau$, is given by the restriction on $\Gamma \vdash e \longrightarrow^* e' : \tau$ 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* $\Gamma \vdash e' \Longmapsto e'' : \tau$ *then* $\Gamma \vdash e[e'/x] \Longmapsto e[e''/x] : \sigma$.

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

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

- $e =_\alpha E_1[e_1]$ *and* $\Gamma \vdash E_1 \mathbin{@} e_1 : \tau' \overset{\Gamma', \overline{\chi}}{\Longrightarrow} \sigma_{ans}$,
- $e_2 =_\alpha e_2[\theta]$ *and* $\Gamma, \Gamma', \chi \vdash e_1 \Rightarrow e_2 : \tau$, *and*
- $\Gamma \vdash E[e_3] \Rightarrow E'[e_3[\theta]] : \sigma_{ans}$ *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 an answer. Note that the restriction on the 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* $\Gamma \vdash e \Rightarrow e_1 \mapsto^* e' : \tau_{ans}$ *then* $\Gamma \vdash e \mapsto^* e_2 \Mapsto e' : \tau_{ans}$.

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

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

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

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

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

$$\frac{\Gamma \vdash E_2 \mathbin{@} e_2 : \sigma \overset{\Gamma'}{\Longrightarrow} \tau_{ans} \quad \Gamma, \Gamma' \vdash e_2 \mapsto e_2' : \sigma}{\Gamma \vdash e \Rightarrow E_2[e_2] \mapsto E_2[e_2'] : \tau_{ans}}$$

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:

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

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

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

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* $\Gamma \vdash e \rightharpoonup^* e_1 \mapsto^* e' : \tau_{ans}$ *then* $\Gamma \vdash e \mapsto^* e_2 \rightharpoonup^* e' : \tau_{ans}$.

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

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

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

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

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* $\Gamma \vdash e \rightarrow^* A : \tau_{ans}$ *then there is an* $\Gamma \vdash A' \rightarrow^* A : \tau_{ans}$ *such that* $\Gamma \vdash e \mapsto^* A' : \tau_{ans}$.

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

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

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

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

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

$$(Force) \qquad \langle x_{\texttt{PtrR}} \mid K \mid [x := \textbf{memo}\, e]H \rangle \mapsto \langle e \mid \textbf{set}\, x; K \mid [x := \bullet]H \rangle$$

$$(Memo) \qquad \langle R \mid \textbf{set}\, x; K \mid [x := \bullet]H \rangle \mapsto \langle R \mid K \mid [x := R]H \rangle$$

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:

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

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, $C[\![e]\!]^\Gamma_\theta(\overline{a})$ 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 $C[\![e]\!]^{\Gamma}_{\theta}(\overline{a})$ 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 $\mathcal{P}[\![P]\!]^{\Gamma}_{\theta}$ 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 $\mathcal{ML}$

$$
\begin{aligned}
A \in \mathit{Answer} &::= H[V] \mid E[\texttt{error}(n)] \\
E \in \mathit{EvalCxt} &::= \Box \mid F[E] \mid B[E] \\
H \in \mathit{ClosCxt} &::= \Box \mid B[H] \\
F \in \mathit{FrameCxt} &::= \texttt{App}\ \Box(\overline{a}) \mid \mathbf{case}\ \Box\ \mathbf{of}\ \texttt{I\#}(x_{\texttt{IntR}}) \to e \mid \mathbf{let}\ x^{\texttt{U}}_{\pi} = \Box\ \mathbf{in}\ e \mid \mathbf{let}\ x^{\texttt{L}}_{\texttt{PtrR}} = \Box\ \mathbf{in}\ E[x_{\texttt{PtrR}}] \\
B \in \mathit{BindCxt} &::= \mathbf{let}\ x^{\texttt{U}}_{\texttt{PtrR}} = R\ \mathbf{in}\ \Box \mid \mathbf{let}\ x^{\texttt{L}}_{\texttt{PtrR}} = e\ \mathbf{in}\ \Box
\end{aligned}
$$

Main reductions:

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

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

$$
\begin{aligned}
(\mathit{Fun}) \quad & \mathbf{let}\ x^{\texttt{U}}_{\texttt{PtrR}} = R\ \mathbf{in}\ E[x_{\texttt{PtrR}}(\overline{a})] \mapsto \mathbf{let}\ x^{\texttt{U}}_{\texttt{PtrR}} = R\ \mathbf{in}\ E[R(\overline{a})] \\
(\mathit{Look}) \quad & \mathbf{let}\ x^{\texttt{U}}_{\texttt{PtrR}} = R\ \mathbf{in}\ E[x_{\texttt{PtrR}}] \mapsto \mathbf{let}\ x^{\texttt{U}}_{\texttt{PtrR}} = R\ \mathbf{in}\ E[R] \\
(\mathit{Memo}) \quad & \mathbf{let}\ x^{\texttt{L}}_{\texttt{PtrR}} = R\ \mathbf{in}\ E[x_{\texttt{PtrR}}] \mapsto \mathbf{let}\ x^{\texttt{U}}_{\texttt{PtrR}} = R\ \mathbf{in}\ E[R]
\end{aligned}
$$

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

$$
\begin{aligned}
(\mathit{LAlloc}) \quad & F[\mathbf{let}\ x^{\texttt{L}}_{\texttt{PtrR}} = e\ \mathbf{in}\ A] \mapsto \mathbf{let}\ x^{\texttt{L}}_{\texttt{PtrR}} = e\ \mathbf{in}\ F[A] \\
(\mathit{SAlloc}) \quad & F[\mathbf{let}\ x^{\texttt{U}}_{\texttt{PtrR}} = R\ \mathbf{in}\ A] \mapsto \mathbf{let}\ x^{\texttt{U}}_{\texttt{PtrR}} = R\ \mathbf{in}\ F[A]
\end{aligned}
$$

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

Translating evaluation contexts to stacks and heaps:

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

$$\mathcal{K}[\![\mathtt{App}\ \square(\overline{a})]\!] \triangleq \mathtt{App}(\overline{a}); \varepsilon \qquad \mathcal{H}[\![\mathtt{App}\ \square(\overline{a})]\!] \triangleq \varepsilon$$

$$\mathcal{K}[\![\mathbf{case}\ \square\ \mathbf{of}\ \mathtt{I\#}(x_{\mathtt{IntR}}) \to e]\!] \triangleq \mathbf{case}\ \mathtt{I\#}(x_{\mathtt{IntR}}) \to e; \varepsilon \qquad \mathcal{H}[\![\mathbf{case}\ \square\ \mathbf{of}\ \mathtt{I\#}(x_{\mathtt{IntR}}) \to e]\!] \triangleq \varepsilon$$

$$\mathcal{K}[\![\mathbf{let}\ x^{\mathsf{U}}_{\pi} = \square\ \mathbf{in}\ e]\!] \triangleq \mathbf{let}\ x_{\pi}\ \mathbf{in}\ e; \varepsilon \qquad \mathcal{K}[\![\mathbf{let}\ x^{\mathsf{U}}_{\pi} = \square\ \mathbf{in}\ e]\!] \triangleq \varepsilon$$

$$\mathcal{K}[\![\mathbf{let}\ x^{\mathsf{L}}_{\mathtt{PtrR}} = \square\ \mathbf{in}\ E[x_{\mathtt{PtrR}}]]\!] \triangleq \mathbf{set}\ x; \mathcal{K}[\![E]\!] \qquad \mathcal{H}[\![\mathbf{let}\ x^{\mathsf{L}}_{\mathtt{PtrR}} = \square\ \mathbf{in}\ E[x_{\mathtt{PtrR}}]]\!] \triangleq \mathcal{H}[\![E]\!][x := \bullet]$$

$$\mathcal{H}[\![\mathbf{let}\ x^{\mathsf{U}}_{\mathtt{PtrR}} = R\ \mathbf{in}\ \square]\!] \triangleq [x := R]$$

$$\mathcal{H}[\![\mathbf{let}\ x^{\mathsf{L}}_{\mathtt{PtrR}} = e\ \mathbf{in}\ \square]\!] \triangleq [x := \mathbf{memo}\ e]$$

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 $\mathbf{let}\ x^{\mathsf{L}}_{\mathtt{PtrR}} = E_2\ \mathbf{in}\ E_1[x_{\mathtt{PtrR}}]$, which proceeds as follows:

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

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

Corollary 5 (Answer Preservation)**.** *If $A \sim m$ then $m \mapsto_F^* A'$. If $e \sim A$, then $e \in$ Answer.*

Proof. By definition of answer 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:

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

- *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 $\langle \mathtt{App}\ (\mathtt{Clos}^n\ P)(\overline{a}) \mid \varepsilon \mid \varepsilon \rangle \mapsto_{PshApp} \langle \mathtt{Clos}^n\ P \mid \mathtt{App}(\overline{a}); \varepsilon \mid \varepsilon \rangle \mapsto_{Apply} \langle P(\overline{a}) \mid \varepsilon \mid \varepsilon \rangle$.
- *Unbox* is similar to *Apply*
- *Look* is

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

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

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

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:

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

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 $\Gamma \vdash e : \tau : \mathsf{TYPE}\ \rho\ \nu$ and $\mathcal{ML}$ expressions is defined as

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

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:

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

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

Lemma 27 (Passive Stability). *$\mathcal{E}_v[\![P]\!]^\Gamma_\theta \triangleq \mathcal{P}[\![P]\!]^\Gamma_\theta$ and $C[\![P']\!]^\Gamma_\theta(\varepsilon) \triangleq \mathcal{P}[\![P']\!]^\Gamma_\theta$ when defined and when P' is not a variable.*

Proof. By induction on the syntax of P. □

Lemma 28 (Answer Preservation). *If $\Gamma \vdash A$ **ans** and $\Gamma \vdash_C A \sim e : \tau$ then $e \in$ Answer. If $\Gamma \vdash_C e \sim A : \tau$, then $e \in$ Answer.*

Proof. The first part follows by induction on the syntax of $\mathcal{IL}$ answers. Of note, in the base cases are where A are references $\mathtt{I\#}^\gamma$ 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 $\mathcal{ML}$ answers are $\mathcal{IL}$ answers. □

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

$$C_\tau[\![\Box]\!]^\Gamma_\theta(\overline{a}) \triangleq \Box\ \overline{a}$$
$$C_\tau[\![E\ a']\!]^\Gamma_\theta(\overline{a}) \triangleq C_\tau[\![E]\!]^\Gamma_\theta(\mathcal{P}[\![a']\!]^\Gamma_\theta, \overline{a})$$
$$C_\tau[\![\mathbf{case}\ E\ \mathbf{of}\ \mathtt{I\#}\ x \to e]\!]^\Gamma_\theta(\overline{a}) \triangleq \mathbf{case}\ C_\tau[\![E]\!]^\Gamma_\theta(\varepsilon)\ \mathbf{of}\ \mathtt{I\#}\ x \to C[\![e]\!]^\Gamma_\theta(\overline{a})$$
$$C_\tau[\![\mathtt{App}\ E]\!]^\Gamma_\theta(\overline{a}) \triangleq \mathtt{App}\ (C_\tau[\![E]\!]^\Gamma_\theta(\varepsilon))(\overline{a})$$
$$C_\tau[\![E\ \phi]\!]^\Gamma_\theta(\overline{a}) \triangleq C_\tau[\![E]\!]^\Gamma_\theta(\overline{a})$$
$$C_\tau[\![\lambda\chi.E]\!]^\Gamma_\theta(\overline{a}) \triangleq C_\tau[\![E]\!]^\Gamma_\theta(\overline{a})$$

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:

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

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

*(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:

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

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

  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

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

- *comm* is, for some ν and ψ

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

  where $Alloc = LAlloc$ if $\psi = \mathsf{L}$ and $Alloc = SAlloc$ if $\psi = \mathsf{U}$.

- *look* is, by Lemma 27, for some ν and ψ

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

  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:

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

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 $\Gamma \vdash_C e'_i \sim e'_m$. □

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. 6.
(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 answers allowed: a necessary restriction to respect the full η-extensionality we're after.

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

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

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

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

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

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

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

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

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

$$[\![a]\!]^{-1} \triangleq a \qquad [\![\mathtt{Int}^{\gamma}]\!]^{-1} \triangleq \mathtt{Int} \qquad [\![{}^{\gamma}\{\tau\}]\!]^{-1} \triangleq [\![\tau]\!]^{-1} \qquad [\![\forall t{:}\kappa.\sigma]\!]^{-1} \triangleq \forall t.[\![\sigma]\!]^{-1} \qquad [\![\sigma \rightsquigarrow \sigma']\!]^{-1}$$

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

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

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

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

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

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

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

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

- $\langle \mathcal{E}_{\mathtt{Eval}^{\mathsf{U}}}[\![e]\!] \mid \varepsilon \mid \varepsilon \rangle \mapsto^* \langle n \mid \varepsilon \mid H \rangle$ from Lemma 25 and Corollary 5.

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

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

□

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-abstraction (of either type), 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,* $[\![V]\!]$ *is passable in* $\mathcal{IL}$.
(2) Given any $\Gamma \vdash e$ **pass** *in the* $\mathcal{IL}$ *sublanguage,* $[\![e]\!]^{-1}$ *is a value in System F.*
(3) Given any $\Gamma \vdash e$ **pass** *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 $\mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Eval}^{\mathsf{L}}$ 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 $\mathsf{TYPE}\ \mathsf{PtrR}\ \mathsf{Eval}^{\mathsf{U}}$. □

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. □

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

(1) $[\![[\![\tau]\!]^{-1}]\!] \triangleq \tau$ *and* $[\![[\![\sigma]\!]^{-1}]\!] \triangleq {}^{\gamma}\{\sigma\}$,
(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* $\Gamma \vdash e$ **pass** *then* $[\![\eta[\![e]\!]^{-1}]\!] =_{\beta_{\{\}}\eta_{\rightsquigarrow}\eta_\forall} \mathsf{Clos}^{\gamma}\ e$.

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 $\mathtt{I\#}^{\gamma}\ n$ 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) $[\![[\![\mathsf{Clos}^{\gamma}\ S]\!]^{-1}]\!] \triangleq [\![\eta[\![S]\!]^{-1}]\!] =_{(4)} \mathsf{Clos}^{\gamma}\ S$
- (3) $\mathsf{App}\ [\![[\![\lambda x{:}\sigma.e]\!]^{-1}]\!] \triangleq \mathsf{App}\ [\![\lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}]\!] \triangleq \mathsf{App}\ (\mathsf{Clos}^{\gamma}\ \lambda x{:}[\![[\![\sigma]\!]^{-1}]\!].[\![[\![e]\!]^{-1}]\!])$
  $=_{(1,2)} \mathsf{App}\ (\mathsf{Clos}^{\gamma}\ \lambda x{:}\sigma.e) =_{\beta_{\{\}}} \lambda x{:}\sigma.e$
- (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) $[\![\eta[\![\lambda x{:}\sigma.e]\!]^{-1}]\!] \triangleq [\![\lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}]\!] \triangleq \mathsf{Clos}^{\gamma}\ \lambda x{:}[\![[\![\sigma]\!]^{-1}]\!].[\![[\![e]\!]^{-1}]\!] =_{(1,2)} \mathsf{Clos}^{\gamma}\ \lambda x{:}\sigma.e$

- (4) $[\![\eta[\![\lambda t{:}\kappa.e]\!]^{-1}]\!] =_{(1,2)} \mathtt{Clos}^{\curlyvee}\ \lambda t{:}\kappa.e$ follows analogously to the previous case. Note this case only applies in call-by-value (de)compilation.
- (4) Given $\Gamma \vdash e : {}^{\curlyvee}\{\sigma \rightsquigarrow \sigma'\}$:

$$[\![\eta[\![\mathtt{App}\ e]\!]^{-1}]\!] \triangleq [\![\lambda x{:}[\![\sigma]\!]^{-1}.[\![e]\!]^{-1}\ x]\!] \triangleq \mathtt{Clos}^{\curlyvee}\ (\lambda x{:}[\![[\![\sigma]\!]^{-1}]\!].\mathtt{App}\ [\![[\![e]\!]^{-1}]\!]\ x)$$
$$=_{(1,2)} \mathtt{Clos}^{\curlyvee}\ (\lambda x{:}\sigma.\mathtt{App}\ e\ x) =_{\eta_{\rightsquigarrow}} \mathtt{Clos}^{\curlyvee}\ (\mathtt{App}\ e)$$

- Given $\Gamma \vdash e : {}^{\curlyvee}\{\forall t{:}\kappa.\sigma\}$, $[\![\eta[\![\mathtt{App}\ e]\!]^{-1}]\!] =_{(1,2)\eta_\forall} \mathtt{App}\ e$ analogously to the previous case. □

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:

- $(\beta_{\{\}})$

$$[\![(\lambda x{:}\tau.e)\ V]\!] \triangleq \mathtt{App}\ (\mathtt{Clos}^{\curlyvee}\ \lambda x{:}[\![\tau]\!].[\![e]\!])\ [\![V]\!]$$
$$=_{\beta_{\{\}}} (\lambda x{:}[\![\tau]\!].[\![e]\!])\ [\![V]\!] =_{\beta_{\rightsquigarrow}} [\![e]\!]\,[[\![V]\!]/x] \triangleq [\![e[V/x]]\!]$$

- $(\beta_\forall)$

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

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

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

- $(\eta_\forall)$

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

- (*name*)

$$[\![(\lambda x{:}\tau.e\ x)\ e']\!] \triangleq \mathtt{App}\ (\mathtt{Clos}^{\curlyvee}\ \lambda x{:}[\![\tau]\!].\mathtt{App}\ [\![e]\!]\ x)\ [\![e']\!]$$
$$=_{\beta_{\{\}}} (\lambda x{:}[\![\tau]\!].\mathtt{App}\ [\![e]\!]\ x)\ [\![e']\!] =_{\eta_{\rightsquigarrow}} \mathtt{App}\ [\![e]\!]\ [\![e']\!] \triangleq [\![e\ e']\!]$$

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

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

  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 $\Gamma \vdash V : {}^{\gamma}\{\sigma \rightsquigarrow \sigma'\}$

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

  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 $\mathtt{App}\ e$ expansion of type $\sigma \rightsquigarrow \sigma'$, which simplify to a known case as follows:

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

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

An analogous derivation is also required for polymorphic application and $\mathtt{App}\ e$ 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 $[\![\Gamma]\!] \vdash [\![e_1]\!] = [\![e_2]\!] : [\![\tau]\!]$.*