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:

\[ \text{twice } f \ x = f (f \ x) \]

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 \texttt{Int} say, but the intermediate language...
can have an unboxed type \texttt{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 \( \text{Int#} \)—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 \( \text{Int#} \), or ban types like \( \text{Int#} \) 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 \( \text{Int#} \) 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 \( \text{F}_{\omega} \), an extension of System F with many different kinds of types for expressing type operators like \( \text{Array} \). In System \( \text{F}_{\omega} \), we are forced to state the kind of types a type variable like \( t \) ranges over. For example, the identity function is elaborated as \( \forall t : \star . t \rightarrow t \), where the kind \( \star \) must somehow specify that \( t \) is represented as a pointer. A type like \( \text{Int#} \), 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 \( \text{plus} \) of type \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \) is regarded as a function of one argument (of type \( \text{Int} \)) that returns a function of one argument (another \( \text{Int} \)), that finally returns an \( \text{Int} \). Such a function can be partially applied. For example \( \text{map} (\text{plus} 1) \) is a function that increments every element in a list; here both \( \text{map} \) and \( \text{plus} \) 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 \( \text{zipWith} \):

\[
\begin{align*}
\text{zipWith } f & \ between \ [\] & = [ ] \\
\text{zipWith } f & \ xs [\] & = [ ] \\
\text{zipWith } f \ (x : xs') \ (y : ys') & = (f \ x \ y) : (\text{zipWith } xs' \ ys')
\end{align*}
\]

The arity of the function \( f \) passed to \( \text{zipWith} \) 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 \( \text{zipWith} \) 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
We thus want a language that supports both lazy and eager evaluation, giving the programmer a choice to compile polymorphic code. Think of these kinds as a coarser-grained description of types, that leave additional details of a definition to be filled in at compile time. 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.\footnote{Previous work \cite{eisenberg2017levity} is titled \textit{Levity Polymorphism}. Yet the paper does not, in our opinion, deliver exactly on the promise in the title. Instead, it describes \textit{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.} 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 \textit{types are calling conventions} \cite{bolingbroke2009representation}. We respectfully disagree, instead claiming that \textit{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 \cite{eisenberg2017levity, pjm91}, arity \cite{bolingbroke2009representation, downen2019kind, marlow2004}, and mixed evaluation strategies \cite{downen2018kinds} in intermediate languages.

Why \textit{kinds}, rather than \textit{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 description, that leave more subtle issues like safety to the finer grain world of types. Better still, making details like...
Kinds are Calling Conventions

<table>
<thead>
<tr>
<th>Types of expressions</th>
<th>( \Gamma \vdash e : \tau )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, x : \tau \vdash x : \tau )</td>
<td>( \text{VAR} )</td>
</tr>
<tr>
<td>( \Gamma \vdash \text{Int}^# e : \text{Int}^# )</td>
<td>( \text{INT}-I )</td>
</tr>
<tr>
<td>( \Gamma \vdash \text{Int}^# e : \text{Int}^# )</td>
<td>( \text{INT-E} )</td>
</tr>
<tr>
<td>( \Gamma, x : \tau \vdash e : \sigma )</td>
<td>( \text{mono-rep} )</td>
</tr>
<tr>
<td>( \Gamma \vdash \lambda x : \tau . e : \tau \leadsto \sigma )</td>
<td>( \text{LAM-I} )</td>
</tr>
<tr>
<td>( \Gamma \vdash e : \tau \leadsto \sigma )</td>
<td>( \text{mono-rep} )</td>
</tr>
<tr>
<td>( \Gamma \vdash e : \tau \leadsto \sigma )</td>
<td>( \text{LAM-P-E} )</td>
</tr>
<tr>
<td>( \Gamma \vdash e : \tau \leadsto \sigma )</td>
<td>( \text{mono-rep} )</td>
</tr>
<tr>
<td>( \Gamma \vdash P : \tau )</td>
<td>( \Gamma \vdash e : \tau \leadsto \sigma )</td>
</tr>
<tr>
<td>( \Gamma \vdash e : \tau \leadsto \sigma )</td>
<td>( \text{LAM-P-E} )</td>
</tr>
<tr>
<td>( \Gamma \vdash Clos^\gamma e : \forall { \tau } )</td>
<td>( \text{CLO-I} )</td>
</tr>
<tr>
<td>( \Gamma \vdash e : \forall \chi, \sigma : \kappa )</td>
<td>( \forall E )</td>
</tr>
<tr>
<td>( \Gamma \vdash \lambda \chi . e : \forall \chi, \sigma )</td>
<td>( \Gamma \vdash [\phi / \chi] \text{poly} )</td>
</tr>
</tbody>
</table>

Polymorphic instantiations: \( \Gamma \vdash [\gamma / g] \text{poly} \)

\( \Gamma \vdash \rho \text{ rep} \)

\( \Gamma \vdash \rho / r \text{ poly} \)

\( \Gamma \vdash \nu \text{ conv} \)

\( \Gamma \vdash [\nu / \nu] \text{ poly} \)

\( \Gamma \vdash \tau : \kappa \)

\( \Gamma \vdash \kappa \text{ kind} \)

<table>
<thead>
<tr>
<th>Types of constants:</th>
<th>( c : \tau )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n : \text{Int}^# )</td>
<td>( \text{error} : \forall r . \forall g . \forall t . \text{Eval}^9 . \text{Int}^# \leadsto t )</td>
</tr>
</tbody>
</table>

Fig. 2. Type system of \( IL \)

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 \( \kappa \) of a type \( \tau \) classifies both the representation and the calling convention of values of type \( \tau \). 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 \( IL \), is an explicitly-typed \( \lambda \)-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, \( \text{Int}^\gamma \) (boxed, where \( \gamma \) refers to the levity, see Section 3.1) and \( \text{Int}^\# \) (unboxed), together with expression forms for explicitly boxing (\( \text{Int}^\# e \)) 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 \( (\tau_1 \leadsto \tau_2) \), rather than the more conventional \( (\tau_1 \rightarrow \tau_2) \), 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 \( \text{Clos}^\gamma e \) and \( \text{App} e \), respectively, are also detailed in Section 3.3.
Kinds of types $\Gamma \vdash \tau : \kappa$

- $\Gamma \vdash \kappa \text{ kind}$
- $\Gamma, t : \kappa \vdash t : \kappa$
- $\Gamma \vdash \text{Int} : \text{TYPE IntR Eval}^\text{Int}$
- $\Gamma \vdash \text{Int} : \text{TYPE PtrR Eval}^\text{Int}$
- $\Gamma \vdash \gamma \text{ lev}$
- $\Gamma \vdash \tau : \text{TYPE} \rho \nu$
- $\Gamma, \chi + \sigma : \text{TYPE} \rho \nu$
- $\Gamma \vdash \text{FORALL}$
- $\Gamma \vdash \{\tau\} : \text{TYPE PtrR Eval}^\text{Int}$
- $\Gamma, \chi + \sigma : \text{TYPE} \rho \nu$
- $\Gamma \vdash \forall \chi, \sigma : \text{TYPE} \rho \nu$
- $\Gamma \vdash \tau_1 : \text{TYPE} \rho_1 \nu_1$
- $\Gamma \vdash \tau_2 : \text{TYPE} \rho_2 \nu_2$
- $\Gamma \vdash \tau_1 \rightsquigarrow \tau_2 : \text{TYPE} \text{PtrR Call}[\rho_1, \text{arity}(\nu_2)]$
- $\Gamma \vdash \text{ARROW}$
- $\Gamma \vdash \tau : \kappa \quad \kappa = \kappa'$
- $\Gamma \vdash \tau : \kappa' \quad \text{K-CONV}$

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

- $\text{arity}(\text{Eval}^\gamma) = \epsilon$
- $\text{arity}(\text{Call}[\alpha]) = \alpha$

Monomorphism restrictions:

- $\Gamma \vdash \tau : \text{TYPE} \rho \nu \vdash \rho \text{ rep}$
- $\Gamma \vdash \tau : \text{TYPE} \rho \nu \vdash \nu \text{ conv}$
- $\Gamma \vdash \tau \text{ mono-rep}$
- $\Gamma \vdash \tau \text{ mono-conv}$

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

- $\Gamma \vdash \rho \text{ rep} \quad \Gamma \vdash \nu \text{ conv}$
- $\Gamma \vdash \kappa \text{ kind}$
- $\Gamma \vdash \text{TYPE} \rho \nu \text{ kind}$
- $\Gamma \vdash \gamma \text{ lev}$
- $\Gamma \vdash \text{L} \text{ lev}$
- $\Gamma \vdash \gamma \text{ lev}$
- $\Gamma, g \vdash g \text{ lev}$
- $\Gamma \vdash \rho \text{ rep}$
- $\Gamma \vdash \text{PtrR} \text{ rep}$
- $\Gamma \vdash \text{IntR} \text{ rep}$
- $\Gamma, r \vdash r \text{ rep}$
- $\Gamma \vdash \gamma \text{ lev}$
- $\Gamma \vdash \text{Eval}^\gamma \text{ conv}$
- $\Gamma \vdash \text{Call}[\alpha] \text{ conv}$
- $\Gamma, \nu \vdash \nu \text{ conv}$
- $\Gamma \vdash \alpha \text{ ari}$
- $\Gamma \vdash \epsilon \text{ ari}$
- $\Gamma \vdash \rho, \alpha \text{ ari}$
- $\Gamma \vdash \text{arity}(\nu) \text{ ari}$

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

• Type abstraction $\lambda \chi. e$ and application $e \phi$. The only unusual thing here is that binders $\chi$ and arguments $\phi$ 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 ($\tau, \gamma, \nu,$ and $\rho$ 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 $\phi$ arguments do not require compilation). Passive expressions include closures $\text{Close} \nu \text{ e}$ as usual, but not functions $\lambda x : \tau. e$ because functions are called instead of evaluated. Types $\tau, \sigma$ include type variables $t$, primitive types $T_\rho$ (of which we supply one, $\text{Int}^\#$), algebraic data types $T_d$ (of which we supply one, $\text{Int}$), polymorphic types $\forall \chi, \sigma$, function types $\tau \rightsquigarrow \sigma$ and a closure type $\nu\{\tau\}$.
3.1 Kinds, representations, calling conventions, and levities

A kind $\kappa$ has the form $\text{TYPE } \rho \nu$, where $\rho$ describes the representation of the type, and $\nu$ 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 $x : \tau : \text{TYPE } \rho \nu$; that is, $x$ is a term variable of type $\tau$, whose kind is $\text{TYPE } \rho \nu$. We now describe what $\rho$ and $\nu$ mean. The representation $\rho$ describes how the value of $x$ is represented at runtime. Referring to Fig. 1, $\rho$ can be:

- $\text{PtrR}$, meaning that $x$ is represented by a pointer into the garbage-collected heap.
- $\text{IntR}$, 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 $\nu$ describes how $x$ may be consumed. Referring to Fig. 1, we see that $\nu$ can be:

- $\text{Eval}^U$, 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.
- $\text{Eval}^L$, 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.
- $\text{Eval}^g$, where $g$ is a levity variable; that is, we support levity polymorphism.
- Call$^a$, meaning that $x$ is a function (not a thunk) with an arity described by $a$. 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 $\rho_i$. Otherwise, the arity will have the form $\rho_1, \ldots, \rho_n, \text{arity}(\nu)$, meaning that $x$ takes at least $n$ arguments, followed by possibly some more arguments given by the arity of another calling convention $\nu$. Call$^a$ is a completely new concept: see Section 3.4.
- $\nu$, a calling-convention variable, which can be bound by a $\forall$; that is, we support polymorphism over calling conventions.

The kind $\text{TYPE } \text{PtrR} \text{Eval}^L$ 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 $\text{TYPE } \text{PtrR} \text{Eval}^U$ and $\text{Eval}^L$ 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 $\text{TYPE } \text{IntR} \text{Eval}^U$. The IntR says that Int# is represented by a machine integer, while the Eval^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 Int$^\gamma$, of kind $\text{TYPE } \text{PtrR} \text{Eval}^\gamma$. This kind specifies that an Int$^\gamma$ is represented by a pointer to a heap-allocated object (hence “boxed”), and that it may be lifted (if $\gamma$ is L) or unlifted (if $\gamma$ 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 $\gamma$ 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

\[
\text{\textbf{plus}} : \text{Int} \sim \text{Int} \sim \text{Int} \quad \text{\textbf{sumFrom}} : \text{Int} \sim \text{Int}
\]

\[
\text{\textbf{plus}} (\text{I\#} \, x) (\text{I\#} \, y) = \text{I\#}(\text{\textbf{plus}} \, x \, y) \quad \text{\textbf{sumFrom}} (\text{I\#} \, 0) = \text{I\#} \, 0
\]

\[
\text{\textbf{sumFrom}} (\text{I\#} \, n) = \text{\textbf{plus}} (\text{I\#} \, n) (\text{\textbf{sumFrom}} (\text{I\#}(\text{\textbf{minus}} \, n) \, 1))
\]

where \text{\textbf{plus}}\# and \text{\textbf{minus}}\# are primitive operations (extending \textit{PrimOp} from Fig. 1), both of type \text{Int\#} \sim \text{Int\#}.

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

\[
\text{\textbf{sumFrom}} : \text{Int} \sim \text{Int} \quad \text{\textbf{sumFrom\#}} : \text{Int} \sim \text{Int\#}
\]

\[
\text{\textbf{sumFrom}} (\text{I\#} \, n) = \text{I\#} \, (\text{\textbf{sumFrom\#}} \, n) \quad \text{\textbf{sumFrom\#}} \, 0 = \text{0}
\]

\[
\text{\textbf{sumFrom\#}} \, n = \text{\textbf{plus}}\# \, n \, (\text{\textbf{sumFrom\#}} \, (\text{\textbf{minus}} \, n) \, 1))
\]

Now, the recursion is done by the more efficient \text{\textbf{sumFrom\#}} function which works directly on machine integers; no boxes are allocated or consumed, and so \text{\textbf{sumFrom\#}} can be compiled with no intermediate allocation. \text{\textbf{sumFrom}} becomes just a \text{\textbf{wrapper}} around \text{\textbf{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 \text{Int\L} from \text{Int\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 \scriptsize\textit{levity} polymorphism (see section Section 3.5).

### 3.3 Lambdas and closures

Consider these two function definitions in, say, OCaml:

\[
f_1 = \lambda x. \, \lambda y. \, \text{\textbf{print}} \, x \, \text{in} \, \text{\textbf{plus}}\# \, x \, y \quad f_2 = \lambda x. \, \text{\textbf{print}} \, x \, \text{in} \, \lambda y. \, \text{\textbf{plus}}\# \, x \, y
\]

These two functions are \(\eta\)-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, \texttt{let z = f1 10 in 20} will just return 20, but \texttt{let z = f2 10 in 20} will first print 10 and then return 20.

For this reason, the OCaml compiler cannot freely \(\eta\)-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]:

\[
f_1, f_2 : \text{Int\#} \rightarrow \text{Int\#} \rightarrow \text{Int\#}
\]

\[
f_1 = \lambda x. \, \lambda y. \, \text{\textbf{let}} \, z = h \, x \, x \, \text{in} \, e \, y \, z \quad f_2 = \lambda x. \, \text{\textbf{let}} \, z = h \, x \, x \, \text{in} \, \lambda y. \, e \, y \, z
\]

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 \(20\) \textit{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 \textit{should} pass the arguments one at a time. Imagine we had the call \(\text{map} \, (f_2 \, 10) \, x\): then we want to share the computation of \((h \, 10 \, 10)\) among all the elements of \(x\).

Unlike \(f_1\), function \(f_2\) can do (potentially expensive) work when applied to one argument.

²Here and elsewhere we reduce notational clutter by using syntactic sugar such as pattern matching.
In this case, we could $\eta$-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 $\text{map}$. We also saw previously that if we have side effects, like in OCaml, then $\eta$-expansion changes semantics, not just efficiency. Even Haskell has the $\text{seq}$ operator, which also makes $\eta$-expansion semantically unsound. In short, unrestricted $\eta$-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 $\eta$-expansion) in the types. In particular, in $\mathcal{IL}$ any expression $e$ whatsoever of type $\tau_1 \rightarrow \tau_2$ can safely be $\eta$-expanded to $\lambda x. e$.

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:

$$f_3 :: \text{Int} \rightarrow \{\text{Int} \rightarrow \text{Int}\}$$

$$f_3 = \lambda x. \text{let } z = h \times x \text{ in } \text{Clos}^\dagger (\lambda y. e \times z)$$

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

How can $f_3$ be called? Clearly we cannot write $(f_3 \ 10 \ 20)$ because the application $(e_1 \ e_2)$ requires that $e_1 : \tau_1 \rightarrow \tau_2$. Instead, $\text{Clos}$ comes with its dual elimination form, $\text{App}$, and we call $f_3$ by writing $(\text{App} (f_3 \ 10 \ 20))$. 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 $\text{Clos}$ and $\text{App}$ are just as you would expect: see rules $\text{ClosI}$ and $\text{ClosE}$ 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, $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ is the type of an arity 2 function. But suppose we have

$$\text{id} : \forall (t : \text{TYPE} \ 	ext{PtrR} \ v). \ t \rightarrow t$$

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

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’sarity (and argument representations). For example:

$$\text{Int} \rightarrow \text{Int} : \text{TYPE} \ 	ext{PtrR} \ \text{Call}^\dagger [\text{Int}]$$

$$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} : \text{TYPE} \ 	ext{PtrR} \ \text{Call}^\dagger [\text{Int}, \text{Int}]$$

$$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} : \text{TYPE} \ 	ext{PtrR} \ \text{Call}^\dagger [\text{Int}, \text{Int}, \text{Int}]$$

The calling-convention component in these kinds is $\text{Call}^\dagger [\alpha]$, where the arity list $\alpha$ 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 $(\tau_1 \rightarrow \tau_2)$ must track these arities. Looking at rule $\text{ARROW}$ in Fig. 3 we see that the kind of $(\tau_1 \rightarrow \tau_2)$ has a calling convention of $\text{Call}^\dagger [p_1, \text{arity}^\dagger (v_2)]$. The first argument has representation $p_1$, the representation of $\tau_1$. The rest of the arguments come from the calling convention of $\tau_2$, via the type-level function $\text{arity}$, whose behavior is defined in Fig. 3. It returns the arity of $\text{Call}^\dagger [\alpha]$; and the empty arity in the case of $\text{Eval}^\dagger$ since no arguments are needed.
to evaluate a non-function. But $v_2$ might also be a variable $v$, and then $\text{arity}(v)$ is stuck; that is why $\text{arity}(v)$ is part of the syntax of $\alpha$ in Fig. 1. Rule $\kappa$-$\text{conv}$ allows calls to $\text{arity}$ 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, which is short-hand for the following definition in $\text{arity}$ whenever desired.

\[ \forall r, g \forall (t : \text{TYPE} \rightarrow \text{Eval}^g). \text{Int}^\# \leadsto t \]

Here, $\text{error}$ 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 $\text{error}$ 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

\[ \text{plus} : \forall g_1, g_2, g_3. \text{Int}^{g_1} \leadsto \text{Int}^{g_2} \leadsto \text{Int}^{g_3} \]

\[ \text{plus} \; g \; (\text{I}^\# \; x) \; (\text{I}^\# \; y) = \text{I}^\# (\text{plus}^\# \; x \; y) \]

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

\[ \text{plus} : \forall g_1, g_2, g_3. \text{Int}^{g_1} \leadsto \text{Int}^{g_2} \leadsto \text{Int}^{g_3} \]

\[ \text{plus} = \lambda g_1, g_2, g_3, \lambda (x': \text{Int}^{g_3}), \lambda (y': \text{Int}^{g_2}). \text{case} \; x' \; \text{of} \; \text{I}^\# \; x \rightarrow \text{case} \; y' \; \text{of} \; \text{I}^\# \; y \rightarrow \text{I}^\# (\text{plus}^\# \; x \; y) \]

Notice that in this definition, the levities $g_i$ of the argument and return types $\text{Int}^{g_i}$ are statically unknown, so we must be able to pattern-match on and return values with unknown levities. Specifically, rule $\text{case}$ in Fig. 2 allows a case-expression to scrutinize an integer of arbitrary levity $y$.

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 $\lambda$), 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 $\mathcal{U}$, the case for handling a thunk is simply dead code.\(^4\)

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

\[ \Gamma \vdash \nu \text{ lev} \quad \Gamma \vdash \tau : \text{TYPE} \rightarrow \text{PtrR} \nu \quad \Gamma \vdash \text{Array}^\# \tau : \text{TYPE} \rightarrow \text{PtrR} \text{Eval}^\# \]

From a representation point of view, an $\text{Array}^\#$ 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 $\text{Array}^\#$ ($\text{Array}^\#$) is a lifted array of pointers, each of which points directly

---

\(^3\)Alternatively, we could require $\text{arity}(v)$ to be fully calculated in the $\text{Arrow}$ kinding rule. This would let us remove $\text{arity}(v)$ from the grammar of arities, but also forces an additional restriction on the formation of types and expressions, specifically $\text{Arrow}$ and $\text{Lam}$-$I$, to rule out $\text{arity}(v)$. 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

\[
\text{revapp } x f = f x
\]

For now, suppose that it returns \( \text{Int} \). What type should \( \text{revapp} \) have? Here are two possibilities:

1. \( \text{revapp} : \forall (t : \text{TYPE PtrR Eval}^1). t \leadsto (t \leadsto \text{Int} \#) \leadsto \text{Int} \#
\)
2. \( \text{revapp} : \forall (t : \text{TYPE IntR Eval}^1). t \leadsto (t \leadsto \text{Int} \#) \leadsto \text{Int} \#
\)

We want to compile a function like \( \text{revapp} \) 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. \( \text{revapp} : \forall (t : \text{TYPE PtrR Eval}^1). t \leadsto (t \leadsto \text{Int} \#) \leadsto \text{Int} \#
\)
4. \( \text{revapp} : \forall (t : \text{TYPE PtrR \text{Call}[\text{IntR}]). t \leadsto (t \leadsto \text{Int} \#) \leadsto \text{Int} \#
\)

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. \( \text{revapp} : \forall v. \forall (t : \text{TYPE PtrR } v). t \leadsto (t \leadsto \text{Int} \#) \leadsto \text{Int} \#
\)

What about the return type of \( f \)? The code for \( \text{revapp} \) 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. \( \text{revapp} : \forall v. \forall (t_1 : \text{TYPE PtrR } v). (t_2 : \text{TYPE r Eval}^1). t_1 \leadsto (t_1 \leadsto t_2) \leadsto t_2
\)

But notice that, unlike the argument type \( t_1 \), \( \text{revapp} \) 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 \( \text{IL} \) 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 \( \text{revapp} \)'s type argument \( t_1 \) to have a representation-polymorphic kind. A second restriction is demonstrated by this function:

\[
\text{twice } f x = f (f x)
\]

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 \( \text{twice} \) either of these types:

1. \( \text{twice} : \forall (t : \text{TYPE PtrR Eval}^1). (t \leadsto t) \leadsto t \leadsto t
\)
2. \( \text{twice} : \forall (t : \text{TYPE PtrR Eval}^1). (t \leadsto t) \leadsto t \leadsto t
\)

but we must choose: unlike \( \text{revapp} \), \( \text{twice} \) 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 \( \Gamma + e : \tau \) must be bound by \( \Gamma \).

\(^7\)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 $\Gamma \vdash \tau \text{ mono-rep}$, defined in Fig. 3 checks that the representation of $\tau$ 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 $\Gamma \vdash \tau \text{ mono-conv}$ 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 IL into a lower level “machine language” ML. In the translation from IL to 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 $\Gamma$: every variable in $\Gamma$ 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 levy, 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 $\sigma$ (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-kindled:

$$\forall r. \forall (t:\text{TYPE} r \text{ Eval}^1). t \sim t' \subseteq \text{ TYPE PtrR Call} [r]$$

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 $\forall t$ in Fig. 2.

3.9 Let bindings

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

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

LET
We can add a similar derived typing rule \( \text{LET-P} \), by composing \( \text{LAM-1} \) and \( \text{LAM-P-E} \), 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 \( \text{Clos/App} \) (namely \( \beta_{(1)} \) and \( \eta_{(1)} \)) and case, \( \mathcal{I}^\# P \) of \( \eta_{\text{Int}} \) are unsurprising, as are the rules for erasable abstractions \( \beta_v \) and \( \eta_v \). More distinctive is \( \eta_{\sim} \), which (as discussed in Section 3.3) allows unrestricted \( \eta \) in either direction for any expression of a function type.

That leaves the reduction rule \( \beta_{\sim} \). As is usual in a call-by-value \( \lambda \)-calculus, only some expressions—called \textit{passable} or \textit{substitutable} expressions, and here denoted by the metavariable \( S \)—can be passed to a function and substituted for its formal parameter by the \( \beta_{\sim} \) rule. Only if the argument of a \( \beta \)-redex is passable does the \( \beta_{\sim} \) 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 \textit{passive} expressions \( P \) to be \textit{passable}. These expression forms include variables (which would always be substituted for passive expressions), constants, and applications of \( \mathcal{I}^\# \) and \( \text{Clos} \). Notice that, thus far, these are all considered values in the call-by-value \( \lambda \)-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 \( \text{Eval}^1 \)—are considered passable, and hence allow \( \beta_{\sim} \) to fire. In contrast, only expressions that are

---

**Fig. 4. Equational theory of \( \mathcal{IL} \)**

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

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \Gamma \vdash \tau : \text{TYPE} \rho \text{ Eval}^1 \quad \Gamma \vdash e : \tau & \quad \Gamma \vdash \tau : \text{TYPE} \rho \text{ Call}[\alpha] \\
\Gamma \vdash P \textbf{ pass} & \quad \Gamma \vdash e \textbf{ pass} & \quad \Gamma \vdash e \textbf{ pass} & \quad \Gamma \vdash e \textbf{ pass}
\end{align*}
\]

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

- \( \beta_{\sim} \) : \( (\lambda x: \tau.e) = S = e[S/x] \) (\( \eta_{\sim} \))
- \( \beta_v \) : \( (\lambda \chi. e) \phi = e[\phi/\chi] \) (\( \eta_v \))
- \( \beta_{(1)} \) : \( \text{App} (\text{Clos}^v e) = e \) (\( \eta_{(1)} \))
- \( \beta_{\text{Int}} \) : \( \text{case } \mathcal{I}^\# P \text{ of } e = e[P/x] \) (\( \eta_{\text{Int}} \))

\[\quad \text{case } e \text{ of } \mathcal{I}^\# x \rightarrow e = e : \text{Int}^v \]

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

That is, the right-hand side of a \text{let} must have statically known representation and calling convention.
passive (P) are values in call-by-value languages. So we can only substitute a variable with an expression of the convention Eval^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 I#^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

\[(\lambda x:\text{Int}^L.e)\ (\text{plus}\ L\ (I^#\ 1)\ (I^#\ 2)) = \beta_\rightarrow e[I L 1 / x]
\]

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

\[(\lambda x:\text{Int}^U.e)\ (\text{plus}\ U\ (I^#U\ 1)\ (I^#U\ 2)) = (\lambda x:\text{Int}^U.e)\ (I^#U\ 3) = \beta_\rightarrow e[I^#U 3 / x]
\]

Here, we cannot apply the \(\beta_\rightarrow\) directly as before, because the argument is not passable: it has the type \(\text{Int}^U : \text{TYPE}\ P\ r\ \text{Eval}^U\), and it is not passable. Instead we must first evaluate the argument; now it becomes passive, and \(\beta_\rightarrow\) can fire.

Of course the substitution done by \(\beta_\rightarrow\) 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 \(\lambda\)-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 levy-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 machine9.

The syntax of \(\mathcal{ML}\) is restricted from the more \(\lambda\)-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 \(\lambda(\overline{e}_a).e\) 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 \(\lambda\)-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].
an arity-1 function returning a closure of an arity-1 function it must be called as \( \text{App} f(1))(2) \). The number of arguments passed at once is explicitly fixed in each \( \lambda \)-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 \( J_L \), so that it is clear from the syntax how to execute programs. In particular, each variable is annotated with its representation \( \pi \), which must be either a pointer \( (\text{PtrR}) \) or integer \( (\text{IntR}) \). 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 \texttt{let} binding is annotated as either eager \( (U) \) or lazy \( (L) \). This controls whether the right-hand side of the \texttt{let} is evaluated first before being bound to the variable, or bound first and evaluated before use.
and evaluated later as needed. Again, this decision about evaluation order must be statically chosen
for each let, so $ML$ does not support polymorphism over these levities.

4.2 The semantics of $ML$

Executing an $ML$ program involves a machine configuration of the form $⟨e \mid K \mid H⟩$, 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
($[x := \text{memo } e]H$). Both call stacks and heaps are conventional, except that a stack may contain
an application of many arguments in a single stack frame, like $\text{App}(\overline{a}); K$. 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
($\text{PshApp}, \text{PshCase}, \text{PshLet}$) and allocating memory ($L\text{Alloc}, V\text{Alloc}$), but note that we include cases
for both lazy bindings ($L\text{Alloc}$) and strict ones ($Psh\text{Let}, V\text{Alloc}$).

Next we have the rules for performing interesting reductions. $\text{Apply}$ resolves the application
of a closure by extracting the function it contains, and $\text{Call}$ calls a known function directly. Note
that $\text{Apply}$ 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,$\text{Call}$ is merely undefined when the arguments don’t match the bound parameters, representing a
type or memory unsafe error. In addition, we have $\text{Move}$ for moving a constant into an appropriate
variable (corresponding to a register) and $\text{Unbox}$ for extracting the contents of a boxed integer.

Finally, we have the rules for handling pointer variables at runtime. $\text{Fun}$ expects a called function
to map to a value. For other pointers, we have to check whether or not it is evaluated, and merely
$\text{Look}$ up values or else $\text{Force}$ thunks.\footnote{Note that this uniform check on pointers $y_{\text{get}}{r_{\text{r}}}^\tau$ is needed to support
levity polymorphism for types like $\text{Int}^\theta$ and $\theta(\tau)$. In a more practical compiler, we could have
specialized code that avoids a check when it is statically known, due to type checking
that $y_{\text{get}}{r_{\text{r}}}^\tau$ must be unlifted, so that the $\text{Look}$ 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.}
When a forced thunk returns a value, it is $\text{Memoized}$ to share
the result on future uses of that pointer.

4.3 Compilation

The compilation transformation from $IL$ to the low-level machine language $ML$ is given in Fig. 6.
The top-level compilation function is $E_v[\llbracket e \rrbracket^\tau_\theta]$, which compiles a typed expression $\Gamma \vdash e : \tau$ where
$\tau$’s convention is $v$. The environment $θ$ is a mapping from $IL$ variables to $ML$ 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 $\text{Int}^\#, \text{Int}^{\gamma}_{\theta}$, and $\overline{\gamma}(\tau)$ can be
evaluated. In contrast, expressions with types like $\tau \leadsto \sigma$ can only be called. Implementing this
distinction is the main role of $E_v[\llbracket e \rrbracket^\tau_\theta]$, which takes into account the calling convention $v$ of $e$. If it is
$\text{Eval}^{\gamma}_\theta$, then we can evaluate the result of $e$ directly by the main compilation translation. Otherwise
if it is $\text{Cali}[\overline{\pi}]$ then $e$ must be called (not evaluated). To make sure that the definition and call
sites of a function match, we always fully $\eta$-expand function expressions when they are defined:
either on the right-hand side of lets 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 $\text{Cali}[r_1, r_2, \text{arity}(v)]$). In any case, we next move to the main
work-horse of compilation, $C[\llbracket e \rrbracket^\tau_\theta(A)$, that produces $ML$ 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 $\text{error}$, we see that
In the following, all equations are tried left-to-right, top-to-bottom.

Top-level eta expansion:
\[ E_\nu [P]_\Gamma = \mathcal{P}[P]_\Gamma \]
\[ E_{\text{Eval}} [e]_\Theta = C [e]_\Theta (e) \]
\[ E_{\text{Call}} [\pi] [e]_\Theta = \lambda (\bar{x}_\pi). C [e]_\Theta (\bar{x}_\pi) \]

Constants:
\[ C[n]_\Theta (e) = n \]
\[ C[\text{error}]_\Theta (a) = \text{error}(a) \]

Variables:
\[ C[x]_\Theta (\bar{e}) = \theta(x) \quad \text{(if } \Gamma \vdash \tau \xrightarrow{\text{conv}} \text{Eval} \psi) \]
\[ C[x]_\Theta (\bar{a}) = (\theta(x))(\bar{a}) \quad \text{(if } \Gamma \vdash \tau \xrightarrow{\text{conv}} \text{Call} [\pi]) \]

Applications (the following equations are tried top-to-bottom):
\[ C[\text{App}] [e]_\Theta (\bar{a}) = \text{App} (C [e]_\Theta (e))(\bar{a}) \]
\[ C[e] [\phi]_\Theta (\bar{a}) = C[e]_\Theta (\bar{a}) \]
\[ C[e] [P]_\Theta (\bar{a}) = C[e]_\Theta (\mathcal{P}[P]_\Gamma, \bar{a}) \quad \text{(if } \mathcal{P}[P]_\Gamma = x_\pi \text{ or } \mathcal{P}[P]_\Gamma = c) \]
\[ C[e] [P]_\Theta (\bar{a}) = \text{let } x_{\text{err}} = \mathcal{P}[P]_\Theta \text{ in } C[e]_\Theta (x_{\text{err}}, \bar{a}) \]
\[ C[e] [e']_\Theta (\bar{a}) = \text{let } \nu \leq \eta = E_\eta [e']_\Theta (\bar{a}) \quad \text{in } C[e]_\Theta (\bar{a}) \quad \text{(if } \Gamma \vdash e' : \tau, \Gamma \vdash \tau \xrightarrow{\text{conv}} \eta, \text{ and } \Gamma \vdash \tau \xrightarrow{\text{conv}} \text{error}) \]

Boxing and unboxing:
\[ C[\#] [f]_\Theta (e) = \mathcal{P}[\#] [f]_\Theta \]
\[ C[\#] [e]_\Theta (e) = \text{let } x_{\text{intr}} = C[e]_\Theta (e) \text{ in } \#(x_{\text{intr}}) \]
\[ C[\text{case } e' \text{ of } \# x \rightarrow e]_\Theta (\bar{a}) = \text{case } C[e']_\Theta (e) \text{ of } \#(x_{\text{intr}}) \rightarrow C[e]_\Theta (x_{\text{intr}}/x) (\bar{a}) \]

Abstractions:
\[ C[\lambda x : \sigma.e]_\Theta (\bar{a}) = C[e]_\Theta (\lambda x : \sigma (\bar{a})) \]
\[ C[\lambda X.e]_\Theta (\bar{a}) = C[e]_\Theta (\lambda X: (\bar{a})) \]
\[ C[\text{Close}^\wr e]_\Theta (e) = \mathcal{P}[\text{Close}^\wr e]_\Theta \]

Passive expressions:
\[ \mathcal{P}[e]_\Theta = c \quad \mathcal{P}[x]_\Theta = \theta(x) \quad \mathcal{P}[\#] [f]_\Theta = \#(\mathcal{P}[f]_\Theta) \]
\[ \mathcal{P}[\#] [P]_\Theta = \mathcal{P}[P]_\Theta \quad \mathcal{P}[\lambda X.P]_\Theta = \mathcal{P}[P]_\Theta \]
\[ \mathcal{P}[\text{Close}^\wr e]_\Theta = \text{Close}^\wr(\text{arity}(\eta)) \quad \mathcal{E}_{\text{Call}^\wr(\text{arity}(\eta))}[e]_\Theta \quad \text{(if } \Gamma \vdash e : \tau \text{ and } \Gamma \vdash \tau \xrightarrow{\text{conv}} \eta) \]

Calculating known representations and conventions:
\[ \Gamma \vdash \tau \xrightarrow{\text{rep}} \pi \quad \text{and} \quad \Gamma \vdash \tau \xrightarrow{\text{conv}} \eta \]
\[ \Gamma \vdash \tau \xrightarrow{\text{rep}} \pi \quad \Gamma \vdash \tau \xrightarrow{\text{conv}} \eta \]
\[ \Gamma \vdash \tau \xrightarrow{\text{rep}} \pi \quad \Gamma \vdash \tau \xrightarrow{\text{conv}} \eta \]

The levi of a known convention:
\[ \nu = \psi (\text{Eval} \psi) \quad \text{and} \quad \text{lev}(\text{Call} [\pi]) = U \]

Fig. 6. Compiling $\mathcal{L}$ to $\mathcal{M}$. 

Kinds are Calling Conventions 17
precisely one argument \( a \) is required. What if the user has written a partial application of \texttt{error}? Such partial applications are always \( \eta \)-expanded to be fully saturated, satisfying the requirement here. Compiling a variable \( x \) looks it up in the environment \( \theta \). However, note that there is different \( \text{ML} \) 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 \( \text{Ca11}[1] \) compiles as \( C[x]_{[\text{PtrR}/x]} \theta (e) = y_{\text{PtrR}}() \), which is a nullary function call, but a boxed integer variable \( x \) with the evaluation convention \( \text{Eval}' \) compiles as \( C[x]_{[\text{PtrR}/x]} \theta (e) = y_{\text{PtrR}} \) which merely looks up the pointer.

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

- If \( \mathcal{P}[P]_{\theta}^\mathcal{E} \) is a variable or constant (after erasure type), then it can be passed directly.
- Otherwise, we name the argument with a \texttt{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{Inv(a)} or \texttt{ClosP \( P \)}, which means the \texttt{let}-binding will always be represented as a pointer into the heap.\(^{11}\)

In the variable case, we do not need to track the levy 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 \textsc{Lam-P-E}.

In the case of a general application \( e \ e' \) to an arbitrary argument that needs to be computed, corresponding to \textsc{Lam-E}, we always generate a \texttt{let} similar to the second case for \( e \ P \). However, for \textsc{Lam-E}, we need to determine the representation, calling convention, and levy of the binding, which could truly be anything. This is why we require the additional restrictions in \textsc{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 \( \text{IL} \) expressions with a known calling convention have a defined compilation to the lower-level, levy-monomorphic \( \text{ML} \). That is, we want to ensure that the compilation \( \mathcal{E}_\eta [e] \) is well-defined for any well-typed closed expression \( \mathit{inv} \ e : \tau : \text{TYPE} \rho \eta \), where the syntax of known calling conventions \( \eta \) is from Fig. 5.

In fact, we allow for a little more levy polymorphism during compilation: \( \mathcal{E}_{\text{Eval}' \eta} [e] \), for an unknown levy \( \eta \), 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 levy polymorphic expressions appearing in strict contexts, such as in the discriminant of a \texttt{case} or argument of \texttt{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 \textit{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 (\(\pi\)) or calling convention (\(\eta\)) for a sub-expression. This appears in Fig. 6 as the highlighted side conditions \(\Gamma \vdash \tau \leadsto \pi\) and \(\Gamma \vdash \tau \leadsto \eta\), respectively. In general, these calculations could fail if the representation of convention in the kind of \(\tau\) are partially unknown—that is, contains free variables. But any closed representation has the form \(\pi\) and any closed convention is equivalent to a \(\eta\). 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 \(\vdash \tau : \text{TYPE} \nu\) then \(E_\nu[e]\) is defined.

This theorem just states when compilation translates a closed IL expression to ML code. We should also expect that compilation preserves the behavior of IL expressions as well. In other words, if an expression is equal to some answer in IL, then executing the compiled code should give the same answer in 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 IL and ML, see Appendices A and B in the supplementary material.

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

1. For any \(\vdash e : \text{Int}\), \(\vdash e = n : \text{Int}\) if and only if \(\langle E_{\text{Eval}}[e]\rangle | e | e \mapsto^* (n | e | H)\).
2. For any \(\vdash e : \text{Int}^r\), \(\vdash e = \text{Int}^r n : \text{Int}^r\) if and only if \(\langle E_{\text{Eval}}[e]\rangle | e | e \mapsto^* (\text{Int}^r | n | e | H)\).

### 5 Dynamic Arity Raising

So far, both the intermediate language IL and lower-level abstract machine 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 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 ML, we can extend its grammar with partial applications of the form \(\text{Clos}^\nu f(\bar{a})\), where \(n > 0\) is the number of remaining arguments expected before the function \(f\) can be called, and \(\bar{a}\) are the arguments applied so far. We can now express the extra rules for dynamic handling a run-time arity mismatch:

\[
\begin{align*}
\text{(Apply)} & \quad \langle \text{Clos}^\nu P(\bar{a}) | \text{App}(\bar{a}') ; K | H \rangle \mapsto \langle P(\bar{a}, \bar{a}') | K | H \rangle & \quad \text{(if } |\bar{a}'| = n) \\
\text{(PApp)} & \quad \langle \text{Clos}^\nu P(\bar{a}) | \text{App}(\bar{a}') ; K | H \rangle \mapsto \langle \text{Clos}^{\nu - |\bar{a}'|} P(\bar{a}, \bar{a}') | K | H \rangle & \quad \text{(if } |\bar{a}'| < n) \\
\text{(OApp)} & \quad \langle \text{Clos}^\nu P(\bar{a}) | \text{App}(\bar{a}', \bar{a}'') ; K | H \rangle \mapsto \langle P(\bar{a}, \bar{a}') | \text{App}(\bar{a}'') ; K | H \rangle & \quad \text{(if } |\bar{a}'| = n, |\bar{a}''| > 0) \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{plus}_1 & : \{\text{Int#} \leadsto \{\text{Int#} \leadsto \text{Int#}\}\} \\
\text{plus}_2 & : \{\text{Int#} \leadsto \text{Int#} \leadsto \text{Int#}\}
\end{align*}
\]

\[
\begin{align*}
\text{plus}_1 = \text{Clos}^l \lambda x:\text{Int#}. \text{Clos}^l \lambda y:\text{Int#}. \text{plus#} x y \\
\text{plus}_2 = \text{Clos}^l \lambda x:\text{Int#}. \lambda y:\text{Int#}. \text{plus#} x y
\end{align*}
\]

\(\text{plus}_1\) is written to receive one argument at a time, whereas \(\text{plus}_2\) expects two arguments at once, and the difference between the two calling conventions is expressed in *both* the expression and its
These arities are made more explicit when compiling to the lower level:

\[
E_{\text{Eval}}[\text{plus}_1] = \text{Clo}^{-1}_1 \lambda(x_{\text{Int}R}). \text{Clo}^{-1}_1 \lambda(y_{\text{Int}R}). \text{plus}^#(x_{\text{Int}R}, y_{\text{Int}R})
\]

\[
E_{\text{Eval}}[\text{plus}_2] = \text{Clo}^{-2}_2 \lambda(x_{\text{Int}R}, y_{\text{Int}R}). \text{plus}^#(x_{\text{Int}R}, y_{\text{Int}R})
\]

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 App plus\(_2\)(1) dynamically creates a partial application object Clo\(^2\)(\(\lambda(x_{\text{Int}R}, y_{\text{Int}R}).\text{plus}^#(x, y))(1), whereas applying plus\(_1\) to two arguments hits the over-application case:

\[
\langle\text{plus}_1 | \text{App}(1, 2); K \mid H \rangle \mapsto_{\text{OApp}} \langle(\lambda(x_{\text{Int}R}). \text{Clo}^{-1}_1 \lambda(y_{\text{Int}R}). \text{plus}^#(x, y))(1) | \text{App}(2); K \mid H \rangle
\]

In other words, it is safe to treat these two expressions interchangeably in our extended ML.

But notice: if we compile only well-typed 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 IL is overly conservative. Programs that are safe to execute in ML would be rejected in 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:

\[
\forall\{\tau \sim \forall\{\sigma\}\} \approx \forall\{\tau \sim \sigma\}
\]

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:

\[
dup_1 : \{\text{Int} \sim \{\text{Int} \sim \text{Int}\}\} \leadsto \text{Int} \sim \text{Int} \leadsto \text{Int} \\
dup_2 : \{\text{Int} \sim \text{Int} \sim \text{Int}\} \leadsto \text{Int} \sim \text{Int} \sim \text{Int}
\]

\[
dup_1 \ f \ x = \text{App}(\text{App} \ f \ x) \ x \\
dup_2 \ f \ x = \text{App} \ f \ x \ x
\]

These compile to the following ML functions:

\[
E_{\text{Call}[\text{Ptr}R, \text{Int}]}[\text{dup}_1] = \lambda(f_{\text{Ptr}R}, x_{\text{Int}R}). \text{App}(\text{App} f_{\text{Ptr}R}(x_{\text{Int}R}))(x_{\text{Int}R})
\]

\[
E_{\text{Call}[\text{Ptr}R, \text{Int}]}[\text{dup}_2] = \lambda(f_{\text{Ptr}R}, x_{\text{Int}R}). \text{App} f_{\text{Ptr}R}(x_{\text{Int}R}, x_{\text{Int}R})
\]

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.
Therefore, the translation coerces the calling convention with a closure type. The only change to

\[ \text{Call-by-name where } \star = \text{TYPE } \text{PtrR} \text{ Eval}^L \]

\[
\begin{align*}
\llbracket \text{Int} \rrbracket &\triangleq \text{Int}^L & \llbracket t \rrbracket &\triangleq t & \llbracket x \rrbracket &\triangleq x & \llbracket n \rrbracket &\triangleq \text{I}^L \uparrow n \\
\llbracket \tau \rightarrow \sigma \rrbracket &\triangleq \llbracket \tau \rrbracket \to \llbracket \sigma \rrbracket & \llbracket \lambda x : \tau . e \rrbracket &\triangleq \text{Clos}^L \llbracket \lambda x : \tau \rrbracket . \llbracket e \rrbracket & \llbracket e' \rrbracket &\triangleq (\text{App} \llbracket e \rrbracket) \llbracket e' \rrbracket \\
\llbracket \forall t . \tau \rrbracket &\triangleq \text{forall}^L . \llbracket \tau \rrbracket & \llbracket \lambda t . e \rrbracket &\triangleq \lambda t . \star . \llbracket e \rrbracket & \llbracket e \tau \rrbracket &\triangleq \llbracket e \rrbracket \llbracket \tau \rrbracket
\end{align*}
\]

\[ \text{Call-by-value where } \star = \text{TYPE } \text{PtrR} \text{ Eval}^U \]

\[
\begin{align*}
\llbracket \text{Int} \rrbracket &\triangleq \text{Int}^U & \llbracket t \rrbracket &\triangleq t & \llbracket x \rrbracket &\triangleq x & \llbracket n \rrbracket &\triangleq \text{I}^U \uparrow n \\
\llbracket \tau \rightarrow \sigma \rrbracket &\triangleq \text{U} \llbracket \tau \rrbracket \to \llbracket \sigma \rrbracket & \llbracket \lambda x : \tau . e \rrbracket &\triangleq \text{Clos}^U \llbracket \lambda x : \tau \rrbracket . \llbracket e \rrbracket & \llbracket e' \rrbracket &\triangleq (\text{App} \llbracket e \rrbracket) \llbracket e' \rrbracket \\
\llbracket \forall t . \tau \rrbracket &\triangleq \text{forall}^U . \llbracket \tau \rrbracket & \llbracket \lambda t . e \rrbracket &\triangleq \lambda t . \star . \llbracket e \rrbracket & \llbracket e \tau \rrbracket &\triangleq (\text{App} \llbracket e \rrbracket) \llbracket e \rrbracket \\
\end{align*}
\]

Fig. 7. Compiling call-by-name and call-by-value System F to 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 IL FROM A HIGHER LEVEL

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 IL.

6.1 Call-by-Name System F to IL

To translate call-by-name System F into IL, we begin by picking a single “uniform” IL kind \(\star\) that is capable of capturing all the types of the source language, namely \(\star = \text{TYPE } \text{PtrR} \text{ Eval}^L\). Each source-language type \(\tau\) will be represented by an IL type \(\llbracket \tau \rrbracket\) 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 \(\text{Int}^L\), which happily has the correct kind. However, even though the function type \(\llbracket \tau \rrbracket \to \llbracket \sigma \rrbracket\) has the correct semantics, it has the wrong kind \(\text{TYPE } \text{PtrR} \text{ Call}^L \text{PtrR}\). 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\(^L\) to mean call-by-name in the high-level equational theory (Fig. 4), but actually
Call-by-name substitutable values:  
\[ V ::= e \]  
Equational axioms (for both call-by-name and call-by-value definitions of \( V \)):  
1. \( \beta_{\rightarrow} \) \( (\lambda x: \tau. e) V = e[V/x] \)  
2. \( \eta_{\rightarrow} \) \( \lambda x: \tau. (V x) = V : \tau \rightarrow \sigma \)  
3. \( \beta_{\psi} \) \( (\lambda t. V) \tau = e[\tau/t] \)  
4. \( \eta_{\psi} \) \( \lambda t. (V t) = V : \forall t. \sigma \)  
5. \( \eta \) \( (\lambda x: \tau. e) e' = e e' \)  

Call-by-value substitutable values:  
\[ V ::= x | \lambda x: \tau. e | \lambda t. e \]  

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

implement call-by-need evaluation for Eval\^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 levy L as call-by-need in IL itself. For example, the call-by-name (\( \beta_{\rightarrow} \)) 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 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 \( \star = \text{TYPE} \text{PtrR} \text{Eval}^U \) for call-by-value evaluation. As before, we can compile source-level types following the invariant that \( \llbracket \tau \rrbracket : \star \), 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 \( \eta \)-equivalent. We need to make sure that this abstraction is still a value even after the polymorphic \( \lambda \) 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 \( \text{Clos}^U (\lambda t: \star. [\llbracket e \rrbracket]) \), which is essential when side effects or non-termination enters the picture. For example, evaluating \( \lambda t. \text{error t n} \) returns immediately in call-by-value System F, but the corresponding \( \lambda t: \star. \text{error} \text{PtrR} U t \) causes an error in 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-IL Compilation

Compiling call-by-name and call-by-value System F into 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 β → 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, \(\text{Int}\). It is easy to see how to generalize the idea to arbitrary user-defined algebraic data types [Graf 2020]. For example, rather than having \(\text{Int}\) built-in, we might define it like this

```plaintext
data \(\text{Int ((g:Lev)} : \text{TYPE \text{PtrR (Eval g)) where}
I\#: \forall (g:Lev) \text{. Int} \sim \text{Int} g
```

Now, the \(\text{Int}\) 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:

```plaintext
data \(\text{List (g:Lev)} (v:Conv) (a: \text{TYPE \text{PtrR (Eval g)) where}
Nil : \text{List} (g \sim \text{a}) a
Cons : a \sim \text{List} (g \sim \text{a}) \rightarrow \text{List} (g \sim \text{a})
```

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:

```plaintext
\begin{align}
\text{sum} & : \forall (g:Lev) \text{. List} (g \sim \text{Eval} g) (\text{Int} g) \sim \text{Int} g \\
\text{sum Nil} & = \text{I\# 0} \\
\text{sum} (\text{Cons} (\text{I\# x}) \text{xs}) & = \text{case sum} \text{xs of} \text{I\# y} \rightarrow \text{I\# (plus\# x y)}
\end{align}
```

This polymorphic definition is possible because the \(\text{sum}\) 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 \(\text{Cons}\) cell, whereas a strict map will first recurse before returning anything.

Because this \(\text{List}\) 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:

```plaintext
\text{Cons } \text{plus\# Nil : \text{List (Call[IntR,IntR]) (Int\# \sim \text{Int\# \sim \text{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—a non-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 naively 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 \((\tau \leadsto \sigma)\), whereas \(\mathcal{IL}\) has just one
arrow \((\tau \leadsto \sigma)\), plus the closure type \(\gamma(\tau)\). The two are inter-convertible: \(\tau \to \sigma \approx \{\tau \leadsto \sigma\}\) and
\(\{\tau\} \approx () \to \tau\). The approach here has a greater economy of concepts, and a nice correspondence
with \(\text{Int}^\#\) and \(\text{Int}^\gamma\). 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 [Wei-rich 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 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 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” (\(\text{Int} = \uparrow \text{Int#}\)) and the unknown function type constructor to a “down shift” (\(\{\sigma \rightsquigarrow \sigma\} = \downarrow \{\sigma \rightsquigarrow \sigma\}\)). Note that for the sake of usability, IL performs implicit polarity conversions of types based on their context. For example, closing over a non-function type like \(\text{Int#}\) implicitly shifts it to a “nullary function” (there written \(\uparrow \text{Int}\)), expressed by the encoding \(\{\text{Int#}\} = \downarrow \uparrow \text{Int#}\).

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


A OPERATIONAL SEMANTICS FOR \( \mathcal{IL} \)

A.1 Notation

We use \( \rightarrow \) to denote the single step relation of an operational semantics, \( \rightarrow^2 \) to denote its reflexive closure, \( \rightarrow^+ \) to denote its transitive closure, and \( \leftarrow^* \) to denote its reflexive-transitive closure. This same notation is used for other reduction arrows.

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

\[ A \in \text{Answer} ::= V \mid E[\text{error} \rho \gamma V] \]
\[ V \in \text{Value} ::= x \bar{\phi} \mid c \bar{\phi} \mid \text{I}#V \mid \text{Clos}^\gamma e \mid \lambda x: \tau. e \mid \lambda \chi. V \]
\[ E \in \text{EvalCxt} ::= \Box \mid e E \mid E e \mid \text{App} E \mid \text{I}#E \mid \text{case} E \text{ of } \text{I}#x \rightarrow e \mid E \phi \mid \lambda \chi. E \]

Note that the definition of \text{value} only differs from \text{passive} in that we don’t allow for \( \beta_\nu \) 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 \( P \rightarrow^* V \). 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:

\[ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \text{TYPE} \rho \text{Call} [\sigma]}{\Gamma \vdash \text{V val}} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau : \text{TYPE} \rho \text{Eval}^L}{\Gamma \vdash e \text{ val}} \]

The decomposition of an expression into an evaluation context surrounding an expression is also defined in part by typing information, written with the judgement \( \Gamma \vdash E @ e : \tau \rightarrow \sigma \), by the following inference rules:

\[ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Box @ e : \tau \rightarrow \tau} \quad \frac{\Gamma \vdash \text{I}# e : \tau \rightarrow \tau}{\Gamma \vdash \text{I}#\gamma e : \tau \rightarrow \tau} \quad \frac{\Gamma \vdash \text{Eval}^L \gamma}{\Gamma \vdash \text{Case} E \text{ of } \text{I}#x \rightarrow e' @ e : \tau \rightarrow \sigma} \]
\[ \frac{\Gamma \vdash \text{Eval}^L \gamma}{\Gamma \vdash \text{Eval}^U \gamma} \quad \frac{\Gamma \vdash e' @ e : \tau}{\Gamma \vdash \text{Eval}^U \gamma \rightarrow \sigma} \]
\[ \frac{\Gamma \vdash e' @ e : \tau}{\Gamma \vdash \text{Eval}^U \gamma \rightarrow \sigma} \quad \frac{\Gamma \vdash e' @ e : \tau}{\Gamma \vdash \text{Eval}^L \gamma \rightarrow \sigma} \quad \frac{\Gamma \vdash e' @ e : \tau}{\Gamma \vdash \text{Eval}^U \gamma \rightarrow \sigma} \]

The (typed) operational stepping relation is written \( \Gamma \vdash e \rightarrow e' : \sigma \), 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):

\[ (\beta_\gamma) \quad \Gamma \vdash (\lambda x: \tau. e) e' \rightarrow e[e'/x] : \sigma \quad \text{ (if } \Gamma \vdash e' \text{ val} \rangle \]
\[ (\beta_\nu) \quad \Gamma \vdash (\lambda \chi. V) \phi \rightarrow V[\phi/\chi] : \sigma[\phi/\chi] \]
\[ (\beta_1) \quad \Gamma \vdash \text{App} (\text{Clos}^\gamma e) \rightarrow e : \sigma \]
\[ (\beta_{\text{Int}}) \quad \Gamma \vdash \text{Case} \text{ I}#V \text{ of } \text{I}#x \rightarrow e \rightarrow e[V/x] : \sigma \]
Additionally, the operational steps are closed under evaluation contexts:

\[
\frac{\Gamma \vdash E @ e : \tau \quad \Gamma \vdash \Gamma' \vdash e \mapsto e' : \tau}{\Gamma \vdash E[e] \mapsto E[e'] : \sigma} \quad \text{compat}
\]

Lemma 1 (Typed Decomposition). If \( \Gamma \vdash E @ e : \tau \vdash' \sigma \quad \Gamma, \Gamma' \vdash e \mapsto e' : \tau \) and \( \Gamma \vdash E[e] : \sigma \).

Proof. By induction on the derivation of the decomposition \( \Gamma \vdash E @ e : \tau \vdash' \sigma \).

Lemma 2 (Unique Decomposition). For every IL expression \( \Gamma \vdash e : \sigma \), either:

1. \( e \in \text{Value} \), or
2. there is a unique \( \Gamma \vdash E @ e' : \tau \vdash' \sigma \) 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 \( \Gamma \vdash e' : \tau \) implies \( \Gamma \vdash e[e''/x] \mapsto e'[e''/x] : \sigma \).
2. If \( \Gamma \vdash [\phi/\chi] \text{ poly} \) 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 \( \Gamma, \chi : \sigma \vdash e : \tau \) 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 \( \Gamma \vdash \gamma \vdash \text{ lev} \) then \( \Gamma[\gamma/g] \vdash e[\gamma/g] : \tau[\gamma/g] \).
4. If \( \Gamma, r \vdash e : \tau \) and \( \Gamma \vdash \rho \vdash \text{ rep} \) then \( \Gamma[\rho/r] \vdash e[\rho/r] : \tau[\rho/r] \).
5. If \( \Gamma, v \vdash e : \tau \) and \( \Gamma \vdash \nu \vdash \text{ conv} \) 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. \( e \in \text{Answer} \), 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

\[
\text{let } x: \tau = e \text{ in } e' \equiv (\lambda x: \tau . e') e
\]

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

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \Gamma \vdash \tau : \text{TYPE\ PtrR\ Call}[a] \\
\Gamma \vdash e \text{ ref} & \\
\Gamma \vdash e \text{ ref} & \\
\Gamma \vdash e \text{ bind} & \\
\Gamma \vdash e \text{ bind} & \\
\end{align*}
\]

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:

\[
\begin{align*}
\Gamma \vdash E @ e : \tau \quad \Gamma \vdash \tau' \rightsquigarrow \sigma & \quad \Gamma \vdash a : \tau' \\
\Gamma \vdash E a @ e : \tau \quad \Gamma \vdash \sigma & \\
\Gamma, x : \tau' + E @ e : \tau & \quad \Gamma \vdash e' : \tau' \quad \Gamma \vdash e' \text{ bind} \quad \Gamma \vdash \tau' \text{ mono-rep} \\
\Gamma \vdash \text{let } x : \tau' = e' \text{ in } E @ e : \tau \quad \Gamma' \quad \Gamma' \vdash \sigma \\
\Gamma \vdash E @ e : \tau \quad \Gamma' \quad \Gamma, x : \tau' + e' : \sigma \quad \Gamma \vdash \tau' : \text{TYPE\ } \rho \text{ EvalL} ^{1} \quad \Gamma \vdash \tau' \text{ mono-rep} \\
\Gamma \vdash \text{let } x : \tau' = E \text{ in } e' @ e : \tau \quad \Gamma' \quad \Gamma' \vdash \sigma \\
\Gamma, x : \tau' + E_1 @ x : \tau' \quad \Gamma' \quad \Gamma \vdash E_2 @ e : \tau \quad \Gamma \vdash \tau' \quad \Gamma \vdash \tau' : \text{TYPE\ PtrR\ EvalL} ^{1} \\
\Gamma \vdash \text{let } x : \tau' = E_2 \text{ in } E_1[x] @ e : \tau \quad \Gamma' \quad \Gamma' \vdash \sigma
\end{align*}
\]
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:

\[
\begin{align*}
\Gamma \vdash H @ V : \tau &\quad \Rightarrow \quad \Gamma' \\
\Gamma \vdash H[V] \ ans &\quad \Rightarrow \quad \sigma \\
\Gamma \vdash E @ error \ \rho \ \gamma \ \tau \ a : \tau &\quad \Rightarrow \quad \sigma \\
\Gamma \vdash E[error \ \rho \ \gamma \ \tau \ a] \ ans &\quad \Rightarrow \quad \sigma
\end{align*}
\]

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

\[
\begin{align*}
(\beta_{\rightarrow}) &\quad \Gamma \vdash (\lambda x: \tau. e) a \mapsto e[a/x] : \sigma \\
(\beta_\gamma) &\quad \Gamma \vdash (\lambda \chi : A) \phi \mapsto A[\phi/\chi] : \sigma[\phi/\chi] \\
(\beta_1) &\quad \Gamma \vdash \text{App} (\text{Close} \ e) \mapsto e : \sigma \\
(\beta_{\text{Int}}) &\quad \Gamma \vdash \text{case} \ I^\gamma \ a \ \text{of} \ I^\# \ x \ \rightarrow \ e \mapsto e[a/x] : \sigma \\
(\text{rename}) &\quad \Gamma \vdash \text{let} \ x: \tau = a \ \text{in} \ e \mapsto e[a/x] : \sigma \\
(\text{name}_1) &\quad \Gamma \vdash I^\gamma e \mapsto \text{let} \ x: \text{Int}^\gamma \ x \ : \text{Int}^\gamma = e \ \text{if} \ e \ \notin \ \text{Arg} \\
(\text{name}) &\quad \Gamma \vdash e' e \mapsto \text{let} \ x: \tau = e \ \text{in} \ e' : \tau \ \text{x} : \sigma \ \text{if} \ (\Gamma \vdash e : \tau \ \text{and} \ e \ \notin \ \text{Arg})
\end{align*}
\]

Because the grammar of evaluation context is simplified to be primarily based on lets—rather than chains of curried function application—we convert more complex applications to an alternative let form, as done by the name and name_1 rules. These let-expressions are interpreted by variable lookup (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 commuting conversions, which push bindings out of the way to bring frames of an evaluation context closer to the root of an answer.

\[
\begin{align*}
\Gamma \vdash E @ x: \tau &\quad \Rightarrow \quad \Gamma' \\
\Gamma \vdash \text{let} \ x: \tau = e \ \text{in} \ E \ [x] &\quad \mapsto \quad \sref \ (\Gamma \vdash e \ \text{ref}) \\
\Gamma \vdash F @ \text{let} \ x: \tau' = e \ \text{in} \ A \ : \tau &\quad \mapsto \quad \sigma \\
\Gamma \vdash \text{let} \ x: \tau' = e \ \text{in} \ A \ ans &\quad \Rightarrow \quad \Gamma' \vdash e \ \text{in} \ A \ : \sigma \ \text{comm} \\
\Gamma \vdash F[\text{let} \ x: \tau' = e \ \text{in} \ A] &\quad \mapsto \quad \text{let} \ x: \tau' = e \ \text{in} \ F[A] : \sigma \\
\Gamma \vdash E @ e : \tau &\quad \Rightarrow \quad \Gamma' \\
\Gamma \vdash E[e] &\quad \mapsto \quad \Gamma', \Gamma' \vdash e' : \tau \ \text{compat}
\end{align*}
\]

**Lemma 7** (Typed Decomposition). If \( \Gamma \vdash E @ e : \tau \ \Rightarrow \ \sigma \) then \( \Gamma, \Gamma' \vdash e : \tau \) and \( \Gamma \vdash E[e] : \sigma \).

**Proof.** By induction on the derivation of the decomposition \( \Gamma \vdash E @ e : \tau \ \Rightarrow \ \sigma \).

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

1. \( e \in \text{Answer} \), or
2. there is a unique \( \Gamma \vdash e' : \tau \ \Rightarrow \ \sigma \) 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 \( e_1 \equiv e_2 \).

**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. \( e \in \text{Answer} \), or
2. \( \vdash e \mapsto e' : \tau \).

Proof. Follows directly from Lemma 8 and compatibility. \( \square \)

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. \( \square \)

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 \( \Gamma \vdash_{\mathcal{UL}} e \sim e' : \tau \) where \( e \) may contain let-expressions but \( e' \) cannot, is defined inductively by the following rules for unwindinglets:

\[
\begin{align*}
\Gamma \vdash_{\mathcal{UL}} e & \sim e_\text{val} : \tau \\
\Gamma \vdash_{\mathcal{UL}} \text{let } x : \tau \mapsto e \sim e' & : \sigma \\
\Gamma \vdash_{\mathcal{UL}} (\text{fun } x : \tau \mapsto e) & \sim (\lambda x : \tau. e') : \sigma \\
\end{align*}
\]

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:

\[
\begin{align*}
\Gamma \vdash_{\mathcal{UL}} e & \sim' : \tau \\
\Gamma \vdash_{\mathcal{NN}} e & \sim e' : \tau \\
\Gamma \vdash_{\mathcal{NN}} (\text{let } x : \tau \mapsto e) & \sim (\text{let } x : \tau \mapsto e') : \sigma \\
\Gamma \vdash_{\mathcal{NN}} (\text{fun } x : \tau \mapsto e) & \sim (\text{fun } x : \tau \mapsto e') : \sigma \\
\end{align*}
\]

Lemma 11 (Context Unwinding). (1) If \( \Gamma \vdash E_\text{e} @ e_\text{e} : \tau \Rightarrow^\Gamma,\Gamma' \Rightarrow_{\mathcal{UL}} e_\text{e}' : \sigma \), and \( \Gamma' \vdash e_\text{e}' : \sigma \), then \( \Gamma \vdash e_\text{e} @ e_\text{e} \Rightarrow_{\mathcal{UL}} E_\text{e}[e_\text{e}'][\theta] : \tau \Rightarrow_{\mathcal{UL}} \sigma \) and

\[
\begin{align*}
\Gamma, \Gamma', \Gamma'' & \vdash e_\text{e} \sim e_\text{e}' : \tau, \text{ and } \Gamma, \Gamma', \Gamma'' & \vdash e_\text{e} \sim e_\text{e} : \tau. \text{ Furthermore, if } \Gamma, \Gamma', \Gamma'' & \vdash e_\text{e} \Rightarrow^* e_\text{e}' : \tau \text{ and } \Gamma, \Gamma', \Gamma'' & \vdash e_\text{e} \Rightarrow^* e_\text{e}' : \tau \text{ such that } \Gamma, \Gamma', \Gamma'' & \vdash e_\text{e} \sim e_\text{e}' : \tau, \text{ then } \Gamma E_\text{e}[e_\text{e}'][\theta] \sim E_\text{e}[e_\text{e}'][\theta] : \tau. \\
\end{align*}
\]

(2) If \( \Gamma \vdash H_\text{e} @ e_\text{e} : \tau \Rightarrow \sigma \) and \( \Gamma \vdash e_\text{e} \Rightarrow_{\mathcal{UL}} \sigma \), then \( \Gamma \vdash H_\text{e} @ e_\text{e} \Rightarrow_{\mathcal{UL}} \sigma \) for some substitution \( \theta = [e_\text{e}/x] \) in the call-by-name \( \mathcal{IL} \) such that \( \Gamma, \Gamma', \Gamma'' \vdash e_\text{e} \sim e_\text{e}' : \tau \).

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 lets):

- The cases for all non-let evaluation contexts follow from the inductive hypothesis.
- \( \Gamma \vdash \text{let } x : \tau' \mapsto e_\text{e} : \tau \Rightarrow^\Gamma,\Gamma',\tau' \Rightarrow_{\mathcal{UL}} e_\text{e}' : \sigma \) because \( \Gamma, x : \tau' \vdash E_\text{e} @ e_\text{e} : \tau \Rightarrow^\Gamma,\Gamma' \Rightarrow_{\mathcal{UL}} e_\text{e}' : \sigma \) and \( \Gamma \vdash e_\text{e}' \Rightarrow_{\mathcal{UL}} e_\text{e}' \) because of the rule of the simulation:
  - \( \text{share} \): We have

\[
\begin{align*}
\Gamma, x : \tau' & \vdash_{\mathcal{UL}} E_\text{e}[e_\text{e}] \sim e_\text{e} : \sigma \\
\Gamma \vdash_{\mathcal{UL}} e_\text{e}' \sim e_\text{e}' : \tau' \\
\Gamma \vdash_{\mathcal{UL}} \text{let } x : \tau' \mapsto e_\text{e} : \tau \Rightarrow_{\mathcal{UL}} \sigma \\
\end{align*}
\]
By the inductive hypothesis, we get that
\[ \Gamma, x : \tau' \vdash E_s : \tau \implies \sigma \quad \Gamma \vdash \lambda x : \tau' . e_s : \tau \quad \Gamma, \Gamma' \vdash \mathcal{U} e_e \sim e_{s_1} : \tau \]
In other words, we reduce to
\[ (\lambda x : \tau' . e_s) \mapsto_{\beta_{\tau'}} e_s'[x/x] \mapsto^* E_s[\theta, e_s'[x/x]] \]
which gives us our new evaluation context and substitution, since evaluation contexts are stable under substitution (Lemma 3). Any further reduction of \( e_e \) and \( e_{s_1} \) preserve the relation by the \textit{copy} rule.

- \textit{copy}: We have \( E_{e_0}[e_e][x/y_1] = E_e[e_e] \) such that
\[
\Gamma, y_1 : \tau' \vdash E_{e_0}[e_{e_0}] \sim e_{s_1} : \sigma \quad \Gamma \vdash \mathcal{U} e_{s_1} \vdash e_{s_1} : \tau' \\
\Gamma \vdash \mathcal{U} \text{ let } x : \tau' = e_{s_1}' \text{ in } E_{e_0}[e_{e_0}][x/y_1] \sim e_{s_1}'[y_1/y_1] : \sigma
\]
By the inductive hypothesis, we get that
\[ \Gamma, y_1 : \tau' \vdash E_{e_0}[e_{e_0}] \sim e_{s_1} : \sigma \quad \Gamma \vdash \mathcal{U} e_{s_1} : \tau' \quad \Gamma \vdash e_{s_1}' \mapsto e_{s_1}' : \tau' \]
In other words, by stability of call-by-name reduction under substitution (Lemma 3), we reduce to \( e_{s_1}[e_{s_1}'[y_1/y_1]] \to^* E_{e_0}[e_{e_0}][\theta, e_{s_1}'[y_1/y_1]] \). Any further reduction of \( e_e \) and \( e_{s_1} \) preserve the relation by the \textit{copy} rule.

- \textit{share}: We have \( \Gamma \vdash E_{e_0}[e_{e_0}][x]@e_e : \tau \implies \sigma \) because \( \Gamma \vdash E_{e_0} @ e_e : \tau \implies \tau' \) and \( \Gamma \vdash \tau' : \text{TYPE } \rho \text{ Eval}^1 \). Only the \textit{share} rule applies, so we have that
\[
\Gamma, x : \tau' \vdash \mathcal{U} e_{s_1} : \tau' \implies \sigma \quad \Gamma \vdash E_{e_0}[e_{e_0}][x] \sim e_{s_1} : \tau' \quad \Gamma \vdash \mathcal{U} \text{ let } x : \tau' = E_{e_0}[e_{e_0}][x] \sim e_{s_1}' : \sigma
\]
By the inductive hypothesis, we get that
\[ \Gamma \vdash e_s \mapsto^* E_{e_0}[e_{e_0}][\theta] : \tau' \quad \Gamma \vdash E_{e_0} @ e_{e_0} : \tau \implies \sigma \quad \Gamma, \Gamma' \vdash \mathcal{U} e_e \sim e_{s_1} : \tau \]
In other words, we reduce to \( (\lambda x : \tau' . e_s) \mapsto^* (\lambda x : \tau' . e_s') E_{e_0}[e_{e_0}][\theta] \). Any further reduction of \( e_e \) and \( e_{s_1} \) preserve the relation by the \textit{share} rule.

- \textit{share}: We have \( \Gamma \vdash E_{e_0} \vdash E_{e_1}[x]@e_e : \tau \implies \sigma \) because \( \Gamma \vdash E_{e_0} \vdash E_{e_0}[e_{e_0}][\theta] \). Any further reduction of \( e_e \) and \( e_{s_1} \) preserve the relation by the \textit{share} rule.

By the inductive hypothesis, we get that
\[
\Gamma, x : \tau' \vdash e_s \mapsto^* E_{e_1}[e_{s_1}][\theta] : \sigma \quad \Gamma \vdash e'_{s_1} \mapsto^* E_{e_2}[e_{s_1}'[\theta]] : \tau' \quad \Gamma \vdash E_{e_0} @ e_{e_0} : \tau \implies \sigma \quad \Gamma \vdash E_{e_0} @ e_{e_0} : \tau \implies \tau' \quad \Gamma \vdash \mathcal{U} e_e \sim e_{s_1}' : \tau' \quad \Gamma, \Gamma' \vdash \mathcal{U} e_e \sim e_{s_1}' : \tau
\]
It follows that \( e_{s_1} \) must be \( x \) and \( x \notin \text{dom}(\theta) \), so because reduction is stable under substitution (Lemma 3) we reduce to
\[
(\lambda x : \tau' . e_s) e'_{s_1} \mapsto_{\beta_{\tau'}} e_s[e'_{s_1}[x] \mapsto^* E_{s_1}[e'_{s_1}] \mapsto^* E_{s_1}[E_{s_2}[e'_{s_1}[\theta]]])
\]
Any further reduction of \( e_e \) and \( e''_{s_1} \) preserve the relation by the \textit{copy} rule, notably expanding the other instances of \( e'_{s_1} \).
Then we know that, given any contexts.

Lemma 12 (Answer Preservation). If \( \Gamma \vdash_\Rightarrow A \sim e : \tau \) then \( e \in \text{Answer} \), and if \( \Gamma \vdash_\Rightarrow e \sim A : \tau \) then \( e \in \text{Answer} \). Likewise, if \( \Gamma \vdash_\sim N \sim e : \tau \) then \( \Gamma \vdash e \mapsto \ast A' : \tau \), and if \( \Gamma \vdash_\sim e \sim A : \tau \) then \( e \in \text{Answer} \).

Proof. The first part about \( \Rightarrow \) follows immediately from Lemma 11 and the fact that answers are closed under substitution. The second part about \( \Rightarrow \) follows by induction on the simulation relation. The remaining statement about \( \sim \) follows by the reductions of the simulation, which are all instances of \( \beta_{\sim} \).

Lemma 13 (Forward Simulation). If \( \Gamma \vdash_\sim e_1 \sim e_2 : \tau \) and \( \Gamma \vdash e_1 \mapsto e'_2 : \tau \) by the call-by-need semantics then \( \Gamma \vdash e_2 \mapsto \ast e'_2 : \tau \) by the call-by-name semantics.

Proof. First consider the cases where an operational step is applied directly to \( e_1 \) (i.e., not \( \text{comp} \)). These cases are as follows:

- \( \beta_{\sim} \), \( \beta_{\pi} \), \( \beta_{(\_)} \), and \( \beta_{\text{int}} \) in call-by-need follow the rule of the same name in call-by-name.
- \( \text{rename} \) is either immediate (due to an application of \( \text{copy} \)) or follows by \( \beta_{\sim} \) (due to an application of \text{share}).
- \( \text{name}_{e_2} \) and \( \text{name} \) follow from the \text{share} unwinding rules followed by a step of \( \text{unname}_{e_2} \) or \( \text{name} \), respectively.
- \( \text{look} \) and \( \text{comm} \) follow immediately when the let is simulated by \( \text{copy} \), and follows by \( \beta_{\sim} \) when simulated by \text{share} or \text{name}.

\( \text{comp} \), follows by Lemma 11 and the fact that reductions are stable under substitution (Lemma 3). In more detail, suppose that

\[
\frac{\Gamma \vdash e @ e : \tau \Rightarrow \sigma, \Gamma' \vdash e \mapsto e'_2 : \tau}{\Gamma \vdash E[e][e'_2] : \sigma}
\]

Then we know that, given any \( \Gamma \vdash_\sim E[e][e_1] \sim e_2 : \sigma \), we have \( \Gamma \vdash e_1 \mapsto \ast E_1[e_1[\theta]] \mapsto \ast E_3[e'_3[\theta]] : \sigma \) such that \( \Gamma \vdash_\sim E_3[e'_3] \sim E_3[e'_3[\theta]] \).

Lemma 14 (Context Rewinding). If \( \Gamma \vdash E @ e : \tau \Rightarrow \sigma \) in the call-by-name \( \mathcal{LL} \) and \( \Gamma \vdash_\Rightarrow e'_2 \sim E_3[e_3] : \sigma \), 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

\[ \Gamma \vdash \eta \quad e_e \sim e_s : \tau' \quad \Gamma, x : \tau' \vdash \eta' = e'_e \sim e'_s : \sigma \]

\[ \Gamma \vdash \eta \text{ let } x: \tau' = e'_e \text{ in } e_e \sim (\lambda x : \tau' e_s) e'_s : \sigma \]

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 \(\Gamma \vdash \tau : \text{TYPE } \rho\ \text{Eval}^U\) (for which the result follows from the inductive hypothesis).

copy: Where

\[ \Gamma \vdash \eta \quad e'_e \sim e'_s : \tau' \quad \Gamma, y_i : \tau' \vdash \eta e_e \sim e_s : \sigma \quad \Gamma \vdash e'_s \ \text{val} \quad \Gamma \vdash e'_s \mapsto e'_s : \tau \]

\[ \Gamma \vdash \eta \text{ let } x: \tau' = e'_e \text{ in } e_e[x/y_i] \sim e_s[e'_s/y_i] : \sigma \]

There are two possibilities for the decomposition of \(e_s[e'_s/y_i]\) into an evaluation context surrounding a sub-expression (let \(\theta = [e'_si/y_0, \theta']\) and \(\theta' = [e'_s/y_1, \ldots ]\):

- Commutation \(e_s[\theta] = E_{s1}[\theta][e_{s1}[\theta]] = E_{s1}[\theta][e_{s1}][\theta]\) where the substitution does not interact with decomposition. In this case, we also have that \(\Gamma, y_i : \tau' \vdash E_{s1} @ e_s : \tau \implies \sigma:\)

- Interference \(e_s[\theta] = E_{s1}[\theta][E_{s2}[\theta][e_{s2}[\theta]]] = E_{s1}[\theta][E_{s2}[\theta][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 \(\theta\). In other words, we have

\[ e_s[\theta] = E_{s1}[\theta'][e'_s] = E_{s1}[\theta'][E_{s2}[e'_s]] \mapsto E_{s1}[\theta'][e'_s] \]

where we know already that \(\Gamma \vdash \eta \quad e'_e \sim e'_s : \tau'\).

\[ \square \]

Lemma 15 (Backward Simulation). If \(\Gamma \vdash \eta \quad e_1 \sim e_2 : \tau \) 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 \(\Gamma_N \vdash e'_1 \sim e'_2 : \tau\).

Proof. Note that any unnaming expansions are either \(\beta_s\) redexes, or preserve the target of the evaluation context (in the case of a strict \(\text{Eval}^U\) 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 \(\Gamma \vdash \eta \quad e_1 \sim e_2 : \tau\)

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:

- \(\beta_v, \beta_\{1\}, \) and \(\beta_{\text{Int}}\) 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 \(\lambda\)-abstraction is still an answer in the case of \(\beta_v\). Note that it is possible that a redex of the form \(F[e]\) (where \(e\ \text{ans}\) might have a \(\text{let}\) inserted in between the frame context and the answer as \(F[B[e]]\). In this case, an additional \(\text{comm}\) reduction is required.

- \(\beta_{\sim;}\) note that in call-by-need we have

\[ \Gamma \vdash (\lambda x : \tau. e) \ e'_e \mapsto_{\text{name}} \text{let } x : \tau. e = e'_e \text{ in } (\lambda x : \tau. e_e) x \mapsto_{\beta_{\sim ;}} \text{let } x : \tau. e = e'_e \text{ in } e_e : \sigma \]

such that the \(\text{copy}\) rule relates this to the call-by-name \(\beta_{\sim ;}\) reduct like so

\[ \Gamma \vdash \eta \quad e'_e \sim e'_s : \tau \quad \Gamma, x : \tau \vdash \eta e_e \sim e_s : \sigma \]

\[ \Gamma \vdash \eta \text{ let } x: \tau. e = e'_e \text{ in } e_e \sim e_s[e'_s/x] \]

\[ \text{copy} \]
This reduction is still possible if an additional let-binding is inserted in the middle of the $\beta$ redex, pushing the application to the argument $a$ inside via \textit{comm}.

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

\begin{itemize}
\item \textbf{share}: Where
\begin{align*}
\Gamma \vdash q e_e & \sim e_s : \tau' \quad \Gamma, x : \tau' \vdash q = e'_e \sim e'_s : \sigma \\
\Gamma \vdash q \text{ let } x : \tau' &= e'_e \text{ in } e_e \sim (\lambda x : \tau' e_s) e'_s : \sigma \\
& \text{ share}
\end{align*}
\end{itemize}

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

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

\begin{itemize}
\item \textbf{copy}: Where
\begin{align*}
\Gamma \vdash q e'_e & \sim e'_s : \tau' \\
\Gamma, y_i : \tau' \vdash q e_e & \sim e_s : \sigma \\
\Gamma & \vdash e'_s \text{ val} \\
\Gamma & \vdash e'_s \mapsto^\ast e'_s : \tau \\
\Gamma & \vdash q \text{ let } x : \tau' = e'_e \text{ in } e_e[x/y_i] \sim e_s[e'_s/y_i] : \sigma \\
& \text{ copy}
\end{align*}
\end{itemize}

There are two possibilities for the decomposition of $e_s[e'_s/y_i]$ into an evaluation context surrounding a sub-expression (let $\theta = [e'_s/y_0, \theta']$ and $\theta' = [e'_s/y_1, \ldots]$):

\begin{itemize}
\item Commutation $e_s[\theta] = E_s[\theta][e_{s1}[\theta]] = E_{s1}[e_{s1}][\theta]$ where the substitution does not interact with decomposition. In this case, we also have that $\Gamma, y_i : \tau' \vdash e_{s1} @ e_{s1} : \tau \Rightarrow \sigma$ before substitution, and so the result follows from the inductive hypothesis.
\item Interference $e_s[\theta] = E_s[\theta][E_{s2}[\theta][e_{s2}[\theta]]] = E_{s1}[y_0][E_{s2}[e_{s2}]/y_0, \theta']$ 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 $\theta$. In other words, we have
\begin{align*}
\Gamma \vdash e'_s & \sim e'_s : \tau' \\
\Gamma & \vdash q \text{ let } x : \tau' = e'_e \text{ in } e_e[x/y_i] \sim e_s[e'_s/y_i] : \sigma \\
& \text{ copy}
\end{align*}
\end{itemize}

\begin{itemize}
\item where we know already that $\Gamma \vdash q e'_e \sim e'_s : \tau'$. Therefore, the reduction in call-by-name is either catching up to \textit{copy}, or if there are any steps remaining afterward, the result follows by the inductive hypothesis. \hfill $\square$
\end{itemize}

\section*{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 $\tau_{\text{ans}}$:

$$\tau_{\text{ans}}, \sigma_{\text{ans}} ::= \text{Int#} \mid \text{Int}^U$$

This restriction prevents the $\eta_{\rightarrow}$ and $\eta_{(1)}$ rules from exposing any underlying computation that needs to be evaluated, and thus, they are unnecessary for evaluating the final answer.

\textbf{Theorem 5.} If $\Gamma \vdash e \mapsto^\ast e' : \tau$ then $\Gamma \vdash e = e' : \tau$.

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

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

$$\Gamma \vdash e \rightarrow e' : \tau$$
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 \( \beta_\eta \) rule to the left-to-right reading of \( \beta_\eta \) rule in Fig. 4. In other words, \( \beta_\eta \) 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 \( \beta_\eta \) rule only applies to values of kind \( \text{Eval}^0 \) 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 \( \Gamma \vdash e \rightarrow^* e_1 : \tau \) and \( \Gamma \vdash e \rightarrow^* e_2 : \tau \) then \( \Gamma \vdash e_1 \rightarrow^* e' : \tau \) and \( \Gamma \vdash e_2 \rightarrow^* e' : \tau \) for some \( e' \).

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

### Corollary 3 (Church-Rosser)

\( \Gamma \vdash e = e' : \tau \) if and only if \( \Gamma \vdash e \rightarrow^* e'' \leftrightarrow^* e' : \tau \).

**Proof.** Follows from Theorem 6.

### Definition 2 (Internal Reduction)

An internal reduction, written \( \Gamma \vdash e \rightarrow e' : \tau \), is any reduction \( \Gamma \vdash e \rightarrow e' : \tau \) such that \( \Gamma \vdash e \not\rightarrow e' : \tau \).

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 \( \Gamma \vdash e \rightarrow e' : \tau \), then \( e \in \text{Answer} \) if \( e' \) is, and \( \Gamma \vdash e : \text{val} \) if \( e' \) is.
2. If \( \Gamma \vdash e \rightarrow e' : \tau_{\text{ans}} \), then \( e' \in \text{Answer} \) if \( e \) is, and \( \Gamma \vdash e' : \text{val} \) if \( e \) is.

**Proof.** By induction on the syntax of answers and derivation of \( \text{val} \), then cases on the possible internal reductions. Note that the restriction on the type in the second part prevents an \( \eta \rightarrow \) or \( \eta \{ \} \) reduction from exposing a non-answer expression.

### Definition 3 (Parallel Reduction)

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

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

### Lemma 18 (Parallel Substitution)

If \( \Gamma, x : \tau \vdash e : \sigma \) and \( \Gamma \vdash e' \Rightarrow e'' : \tau \) then \( \Gamma \vdash e[x/x] \Rightarrow e''[x/x] : \sigma \).

**Proof.** By induction on the derivation of \( \Gamma, x : \tau \vdash e : \sigma \).

### Lemma 19 (Internal Decomposition)

If \( \Gamma \vdash e \Rightarrow E_2[e_2] : \sigma_{\text{ans}} \) and \( \Gamma \vdash E_2 \circ e_2 : \tau \), then there is an \( E_1 \) and a \( \theta = [\phi/\chi] \) and \( \tau' \) such that

- \( e =_\theta E_1[e_1] \) and \( \Gamma \vdash E_1 @ e_1 : \tau' \Rightarrow \sigma_{\text{ans}} \),
- \( e_2 =_\theta E_2[\theta] \) and \( \Gamma', \chi \vdash e_2 : \tau', \) and
- \( \Gamma \vdash E[e_3] \Rightarrow E'[e_3[\theta]] : \sigma_{\text{ans}} \) for any \( \Gamma, \Gamma', \chi \vdash e_3 : \tau \).
Proof. By induction on the syntax of evaluation contexts, and inversion on the possible internal reductions. The substitutions $\theta$ are possible due to internal applications of $\beta\eta$ to an abstraction $\lambda\chi.e$ with $e$ not an answer. Note that the restriction on the answer type required to prevent an internal $\eta\rightarrow$ or $\eta\rightarrow_1$ reduction from exposing a deeper evaluation context inside the body of a $\lambda$-abstraction or closure. As such, each application of these two $\eta$ rules must either occur outside the eye of the evaluation context, or inside a $\beta\rightarrow$ or $\beta\rightarrow_1$ redex. In the latter case, such $\eta$ reductions mimic $\beta$ operational steps, therefore making them non-internal and ruling them out. □

Lemma 20 (Standard Preplementation). If $\Gamma \vdash e \Rightarrow e_1 \mapsto e' : \tau_{\text{ans}}$ then $\Gamma \vdash e \mapsto e_2 \Rightarrow e' : \tau_{\text{ans}}$.

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

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

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

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

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

$$\Gamma \vdash E_2 \odot e_2 : \sigma \Rightarrow^* \tau_{\text{ans}} \quad \Gamma, \Gamma' \vdash e_2 \Rightarrow e_2' : \sigma$$

$$\Gamma \vdash e \Rightarrow E_2[e_2] \Rightarrow E_2[e_2'] : \tau_{\text{ans}}$$

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

$$\Gamma, \Gamma', \chi \vdash e_1 \Rightarrow e_3 \Rightarrow e_3' : \tau_{\text{ans}} \quad \Gamma, \Gamma \vdash e_1[\theta] \Rightarrow e_2 \Rightarrow e_2' : \tau_{\text{ans}}$$

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

$$\Gamma, \Gamma \vdash e_1 \Rightarrow e_4 \Rightarrow e_4' : \tau_{\text{ans}} \quad \Gamma \vdash E_1[e_1] \Rightarrow E_1[e_4] \Rightarrow E_2[e_3'[\theta]] =_\alpha E_2[e_3'] : \tau_{\text{ans}}$$

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

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

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

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

$$e_1 \Rightarrow^* e_n \text{ if } e_1 \Rightarrow^* e_1 \Rightarrow^* \ldots \Rightarrow^* e_{n-1} \Rightarrow^* e_n$$

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

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

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

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

B CORRECTNESS OF $IL$-TO-$ML$ COMPILATION

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

$\langle \text{Force} \rangle \quad \langle x_{\text{ptr}} : R \mid K \mid [x := \text{memo } e | H] \rangle \mapsto \langle e \mid \text{set } x ; K \mid [x := \bullet | H] \rangle$

$\langle \text{Memo} \rangle \quad \langle R \mid \text{set } x ; K \mid [x := \bullet | H] \rangle \mapsto \langle R \mid K \mid [x := R | H] \rangle$

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:

$\pi \overset{\text{rep}}{\sim} x_\pi \quad n \overset{\text{rep}}{\sim} \text{IntR} \quad \text{error} \overset{\text{rep}}{\sim} \text{PtrR}$

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

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

Note that the main compilation of expressions, $C[e]_\theta$ 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 $ML$ 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 $\tau \overset{\text{conv-lp}}{\sim} \nu$—as defined by the following rules for $\Gamma \vdash \tau \overset{\text{conv-lp}}{\sim} \eta$:

$\Gamma \vdash \tau \overset{\text{conv-lp}}{\sim} \eta \quad \Gamma \vdash \tau : \text{TYPE } \rho \overset{\text{Eval}^I}{\sim} \Gamma \vdash \tau \overset{\text{conv-lp}}{\sim} \text{Eval}^I$

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

Lemma 23 (Open Compilation). $E \overset{\varphi}{\sim} e_\theta$ is defined if:

1. $\Gamma \vdash e : \tau$ is derivable for some type $\tau$.
2. $\Gamma \vdash \tau \overset{\text{conv-lp}}{\sim} \nu$ for some $\nu$, and
3. $\Gamma \vdash \theta \text{ mono-rep}$. 

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

1. $\Gamma \vdash e : \tau$ is derivable for some type $\tau$,
2. $\Gamma \vdash \tau \xrightarrow{\text{conv}} \nu$ for some $\nu$, such that $|\mathcal{A}| = |\text{arity}(\nu)|$, and $a_i \xrightarrow{\text{rep}} \pi_i$ for each $\pi_i \in \text{arity}(\nu)$, and
3. $\Gamma \vdash \theta$ $\text{mono-rep}$.

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

1. If $\vdash \rho \text{ rep}$, then $\rho = \pi$ for some $\pi$.
2. If $\vdash \gamma \text{ lev}$, then $\gamma = \psi$ for some $\psi$.
3. If $\vdash \nu \text{ conv}$, then either $\nu = \eta$ for some $\eta$. □

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

$A \in \text{Answer} ::= H[V] \mid E[\text{error}(n)]$

$E \in \text{EvalCxt} ::= \Box \mid F[E] \mid B[E]$

$H \in \text{ClosCxt} ::= \Box \mid B[H]$

$F \in \text{FrameCxt} ::= \text{App} \circ \Box(\mathcal{A}) \mid \text{case} \Box \text{of} I#(x_{\text{Intr}}) \rightarrow e \mid \text{let} x_\pi^U = \Box \text{in} e \mid \text{let} x_{\text{PtrR}}^U = \Box \text{in} E[x_{\text{PtrR}}]$

$B \in \text{BindCxt} ::= \text{let} x_{\text{PtrR}}^U = R \text{in} \Box \mid \text{let} x_{\text{PtrR}}^L = e \text{in} \Box$

Main reductions:

(Call) $(\lambda(x_\pi).e)(\mathcal{A}) \mapsto e[\mathcal{A}/x_\pi]$

(Apply) $\text{App}(\text{Clos}^n P)(\mathcal{A}) \mapsto P(\mathcal{A})$ (if $|\mathcal{A}| = n$)

(Move) $\text{let} x_\pi^U = c \text{ in } e \mapsto e[c/x_\pi]$

(Unbox) $\text{case } I#(a) \text{ of } I#(x_{\text{Intr}}) \rightarrow e \mapsto e[a/x_{\text{Intr}}]$

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

(Fun) $\text{let} x_{\text{PtrR}}^U = R \text{in } E[x_{\text{PtrR}}(\mathcal{A})] \mapsto \text{let} x_{\text{PtrR}}^U = R \text{in } E[R(\mathcal{A})]$

(Look) $\text{let} x_{\text{PtrR}}^L = R \text{in } E[x_{\text{PtrR}}] \mapsto \text{let} x_{\text{PtrR}}^U = R \text{in } E[R]$

(Memo) $\text{let} x_{\text{PtrR}}^L = R \text{in } E[x_{\text{PtrR}}] \mapsto \text{let} x_{\text{PtrR}}^U = R \text{in } E[R]$

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

(Alloc) $F[\text{let} x_{\text{PtrR}}^U = e \text{ in } A] \mapsto \text{let} x_{\text{PtrR}}^L = e \text{ in } F[A]$

(SAlloc) $F[\text{let} x_{\text{PtrR}}^U = R \text{in } A] \mapsto \text{let} x_{\text{PtrR}}^U = R \text{ in } F[A]$

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

Translating evaluation contexts to stacks and heaps:

$\mathcal{K}[\Box] \triangleq \epsilon$

$\mathcal{K}[F(E)] \triangleq \mathcal{K}[E] \circ \mathcal{K}[F]$

$\mathcal{K}[B(E)] \triangleq \mathcal{K}[E]$

$\mathcal{H}[\Box] \triangleq \epsilon$

$\mathcal{H}[F(E)] \triangleq \mathcal{H}[E] \circ \mathcal{H}[F]$

$\mathcal{H}[B(E)] \triangleq \mathcal{H}[E] \circ \mathcal{H}[E]$
\[ K[\text{App } \square(x)] \triangleq \text{App}(\square \varepsilon) \quad H[\text{App } \square(x)] \triangleq \varepsilon \]
\[ K[\text{case } \square \text{ of } I#(x_{\text{int}r}) \to e] \triangleq \text{case } I#(x_{\text{int}r}) \to e \quad H[\text{case } \square \text{ of } I#(x_{\text{int}r}) \to e] \triangleq \varepsilon \]
\[ K[\text{let } x = \square \text{ in } e] \triangleq \text{let } x \in e \quad H[\text{let } x = \square \text{ in } e] \triangleq \varepsilon \]
\[ K[\text{let } x_{\text{ptr}r} = \square \text{ in } E[x_{\text{ptr}r}]] \triangleq \text{set } x; K[E] \quad H[\text{let } x_{\text{ptr}r} = \square \text{ in } E[x_{\text{ptr}r}]] \triangleq H[E][x := \bullet] \]
\[ H[\text{let } x_{\text{ptr}r} = R \text{ in } \square] \triangleq [x := R] \quad H[\text{let } x_{\text{ptr}r} = e \text{ in } \square] \triangleq [x := \text{memo } e] \]

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

**Lemma 24.** \( \langle E[e] \mid K \mid H \rangle \rightarrow_F^* \langle e \mid K \mid H \rangle \)

**Proof.** By induction on the context of \( E \). The most interesting case is for an evaluation context of the form \( \text{let } x_{\text{ptr}r} = E_2 \text{ in } E_1[x_{\text{ptr}r}] \), which proceeds as follows:
\[ \langle \text{let } x_{\text{ptr}r} = E_2[e] \text{ in } E_1[x_{\text{ptr}r}] \mid K \mid H \rangle \rightarrow_{\text{Alloc}} \langle E_1[x_{\text{ptr}r}] \mid K \mid [x := \text{memo } E_2[e]]H \rangle \]
\[ \rightarrow_{1H} \langle E_2[e] \mid K[E_1] \circ K \mid H[E_1] \circ [x := \text{memo } E_2[e]]H \rangle \]
\[ \rightarrow_{\text{Force}} \langle E_2[e] \mid K[E_1] \circ K \mid H[E_1] \circ [x := \bullet H] \rangle \]
\[ \rightarrow_{1H} \langle e \mid K[E_2] \circ K \mid H[E_2] \circ H[E_1] \circ [x := \bullet H] \rangle \]

And note that the corresponding stack is \( K[\text{let } x_{\text{ptr}r} = E_2 \text{ in } E_1[x_{\text{ptr}r}]] = K[E_2] \circ K[E_1] \) and heap is \( H[\text{let } x_{\text{ptr}r} = E_2 \text{ in } E_1[x_{\text{ptr}r}]] = H[E_2] \circ H[E_1] \circ [x := \bullet] \)

**Corollary 5** (Answer Preservation). If \( A \sim m \) then \( m \rightarrow F^* A' \). If \( e \sim A \), then \( e \in \text{Answer} \).

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

**Definition 5** (Machine Simulation). The simulation relation between \( \mathcal{ML} \) expressions and machine states is
\[ e \sim m \text{ iff } \langle e \mid e \mid e \rangle \rightarrow_F^* m \]

**Lemma 25** (Forward Simulation).

1. If \( e \rightarrow e' \) by LAlloc, or SAlloc and \( e \sim m \) then \( e' \sim m \).
2. If \( e \rightarrow* e' \) by any other step and \( e \sim m \) then \( m \rightarrow_F^* m'' \) and \( e' \sim m'' \).

Therefore, \( e \rightarrow* e' \) and \( e \sim m \) then \( m \rightarrow_F^* m' \) and \( e' \sim m' \) for some \( m' \).

**Proof.** First, consider the cases where \( e \rightarrow e' \) 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.

- \( \text{LAlloc} \) is, due Lemma 24 and the fact that the heap is unordered:
\[ \langle F[\text{let } x_{\text{ptr}r} = e \text{ in } A] \mid e \mid e \rangle \rightarrow_F^* \langle \text{let } x_{\text{ptr}r} = e \text{ in } A \mid K[F] \mid H[F] \rangle \]
\[ \rightarrow_{\text{LAlloc}} \langle A \mid K[F] \mid [x := \text{memo } e]H[F] \rangle \]
\[ \leftarrow_F^* \langle F[A] \mid e \mid [x := \text{memo } e] \rangle \]
\[ \leftarrow_F \langle \text{let } x_{\text{ptr}r} = e \text{ in } F[A] \mid e \mid e \rangle \]

- \( \text{SAlloc} \) is similar to \( \text{LAlloc} \).

- Call is \( \langle (\lambda(x).e)(\overline{a}) \mid e \mid e \rangle \rightarrow_{\text{Call}} \langle e[x := \overline{a}] \mid e \mid e \rangle \).

- Move is similar to Call.
• \textit{Apply} is $\langle \text{App} (\text{Clos}^n P)(\overline{a}) \mid e \mid e \rangle \mapsto_{P \text{shApp}} \langle \text{Clos}^n P \mid \text{App}(\overline{a}) \rangle \mid e \mid e \rangle \mapsto_{\text{App}} \langle P(\overline{a}) \mid e \mid e \rangle$.

• \textit{Unbox} is similar to \textit{Apply}

• \textit{Look} is

\[
\langle \text{let } x_{PtrR}^U = R \text{ in } E[x_{PtrR}] \mid K \mid H \rangle \mapsto^*_{P} \langle E[x_{PtrR}] \mid K \mid [x := R]H \rangle
\]

\[
\text{↓}_P \langle x_{PtrR} \mid K \mid \mathcal{K}[E] \circ [x := R]H \rangle
\]

\[
\text{↓}_P \langle E[R] \mid K \mid [x := R]H \rangle
\]

\[
\text{↓}_P \langle \text{let } x_{PtrR}^U = R \text{ in } E[R] \mid K \mid H \rangle
\]

• \textit{Fun} is similar.

• \textit{Memo} is

\[
\langle \text{let } x_{PtrR}^L = R \text{ in } E[x_{PtrR}] \mid K \mid H \rangle \mapsto_{F} \langle E[x_{PtrR}] \mid K \mid [x := \text{memo } R]H \rangle
\]

\[
\text{↓}_F \langle x_{PtrR} \mid \mathcal{K}[E] \circ K \mid \mathcal{H}[E] \circ [x := \text{memo } R]H \rangle
\]

\[
\text{↓}_F \langle R \mid \text{set } x; \mathcal{K}[E] \circ K \mid \mathcal{H}[E] \circ [x := \text{•}]H \rangle
\]

\[
\text{↓}_F \langle \text{set } x; \mathcal{K}[E] \circ K \mid \mathcal{H}[E] \circ [x := \text{•}]H \rangle
\]

\[
\text{↓}_F \langle \text{let } x_{PtrR}^L = R \text{ in } E[x_{PtrR}] \mid K \mid H \rangle
\]

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:

\[
\langle E[e] \mid K \mid H \rangle \mapsto_{P} m \mapsto_{P} \langle e \mid \mathcal{K}[E] \circ K \mid \mathcal{H}[E] \circ H \rangle
\]

\[
\text{↓}_P \langle \text{let } x_{PtrR}^L = R \text{ in } E[x_{PtrR}] \mid K \mid H \rangle
\]

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. \hfill \relax

\textbf{Lemma 26} (Backward Simulation). \textit{If }$m \mapsto^* m'$ \textit{and }$e \sim m$ \textit{then }$e' \sim m'$ \textit{and }$e \mapsto^* e'$ \textit{for some }$e'$. \hfill \relax

\textbf{Proof.} Note that if $m \mapsto^* m'$ then $e \sim m'$ directly. Otherwise, if $m \mapsto_R m'$ then $e \mapsto^*_{\text{Alloc,SAloc}} e'' \mapsto e'$ and by forward simulation Lemma 25 and determinism of the abstract machine, we know that $e' \sim m'$. \hfill \relax

\textbf{B.4 Bisimulation between }IL\textbf{ and }ML\textbf{.}

Here, we relate the environment-based call-by-need operational semantics for $IL$ with the operational semantics for $ML$, 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 lists as applied $\lambda$-abstractions.

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

\[
\Gamma \vdash_G e \sim e' : \tau \text{ iff } e' \mapsto_{\text{Unlift}}^* C[e]^F_{\theta}(\bar{a})
\]

\text{for some }$\theta$ \text{ and }$\bar{a}$ \text{ such that }$\Gamma \vdash \theta \text{ mono-rep}$ \text{ and }$\bar{a} \sim_{\text{rep}} \text{arity}(\nu)$, \text{where the }$\text{Unlift}$ \text{ reduction is:}

\[
\text{(Unlift)} \quad \text{let } x_{PtrR}^L = R \text{ in } e \mapsto \text{let } x_{PtrR}^U = R \text{ in } e \]
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). $E \gamma [P]_0^\Gamma \triangleq \mathcal{P}[P]_0^\Gamma$ and $C \gamma [P']_0^\Gamma (e) \triangleq \mathcal{P}[P']_0^\Gamma$ when defined and when $P'$ is not a variable.

**Proof.** By induction on the syntax of $P$. □

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

**Proof.** The first part follows by induction on the syntax of $\mathcal{IL}$ answers. Of note, in the base cases are where $A$ is references $\text{I}#^y \text{arg}$ or $\text{Clos}^y e$ (which have a convention of $\text{Eval}^y$ and cannot be applied to any arguments) or a constant $c \notin \phi$ (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 $\tau$) to $\mathcal{ML}$ evaluation contexts is written as $C_{\tau} [E]_0^\Gamma (\bar{a})$. The definition for most cases, where strictness is obvious from the syntax, are defined as follows:

$$C_{\tau} \square \Gamma (\bar{a}) \triangleq \square \bar{a}$$

$$C_{\tau} [E \, a']_0^\Gamma (\bar{a}) \triangleq C_{\tau} [E]_0^\Gamma (\mathcal{P}[a']_0^\Gamma, \bar{a})$$

$$C_{\tau} \text{case } E \, \text{of } \text{I}# x \to e \, \Gamma (\bar{a}) \triangleq \text{case } C_{\tau} [E]_0^\Gamma (e) \, \text{of } \text{I}# x \to C_{\tau} e \Gamma (\bar{a})$$

$$C_{\tau} \text{App } E \, \Gamma (\bar{a}) \triangleq \text{App} \, (C_{\tau} [E]_0^\Gamma (e)) (\bar{a})$$

$$C_{\tau} [E \, \phi]_0^\Gamma (\bar{a}) \triangleq C_{\tau} [E]_0^\Gamma (\bar{a})$$

$$C_{\tau} \lambda x. E \, \Gamma (\bar{a}) \triangleq C_{\tau} [E]_0^\Gamma (\bar{a})$$

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:

$$C_{\tau} \text{let } x: \sigma = R \in E \, \Gamma (\bar{a}) \triangleq \text{let } x^\sigma_{\text{ptrR}} \in C_{\tau} [E]_0^\Gamma,x: \sigma_{\text{[x]}_0^\sigma} (\bar{a})$$

$$C_{\tau} \text{let } x: \sigma = e \in E \, \Gamma (\bar{a}) \triangleq \text{let } x^\sigma_{\text{lev(\eta)}} \in C_{\tau} [E]_0^\Gamma,x: \sigma_{\text{[x]}_0^\sigma} (\bar{a})$$

$(\text{if } \Gamma \vdash _e \text{ bind and } \Gamma \vdash _\sigma \overset{\text{conv}}{\rightsquigarrow} _\eta)$

$$C_{\tau} \text{let } x: \sigma = E \, e \, \Gamma (\bar{a}) \triangleq \text{let } x^\sigma_{\text{pi}} \in C_{\tau} [E]_0^\Gamma,x: \sigma_{\text{[x]}_0^\sigma} (\bar{a})$$

$(\text{if } \Gamma \vdash _\sigma \overset{\text{conv}}{\rightsquigarrow} \text{Eval}^U \text{ and } \Gamma \vdash _\sigma \overset{\text{rep}}{\rightsquigarrow} _\pi)$

$$C_{\tau} \text{let } x: \sigma = E_2 \, E_1 \, x \, \Gamma (\bar{a}) \triangleq \text{let } x^\sigma_{\text{ptrR}} \in C_{\tau} [E]_0^\Gamma,x: \sigma_{\text{[x]}_0^\sigma} (\bar{a})$$

$(\text{if } \Gamma \vdash _\sigma \overset{\text{conv}}{\rightsquigarrow} \text{Eval}^L)$

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 $\Gamma \vdash E \, @ \, e : \tau \overset{\Gamma'}{\Rightarrow} \sigma$ then

$$C [E[e]]_0^\Gamma (\bar{a}) \triangleq C_{\tau} [\text{Eval}^\Gamma (e)]_0^\Gamma (\bar{a})$$

**Proof.** By induction on the derivation of $\Gamma \vdash E \, @ \, e : \tau \overset{\Gamma'}{\Rightarrow} \sigma$. □
Lemma 30 (Instantiation).  (1) For any $\Gamma, \chi : t \vdash e : \sigma$ and $\Gamma \vdash a' : \tau$, renaming an argument for a variable commutes with compilation: $C[e[a'/x]]^\Gamma_\theta(a) \triangleq C[e]^\Gamma_{\{P[a'/x]\theta\}}(a)\theta$.

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

Proof. By induction on the syntax of $e$. $\Box$

Lemma 31 (Forward Simulation). For any $IL$ expression $e_i$ and $ML$ expression $e_m$,

(1) If $\Gamma \vdash e_i \sim e_m : \tau$ and $\Gamma \vdash e_i \leftrightarrow e'_i : \tau$ by a $\beta_\rightarrow$, $\beta_i$, rename, name, or $\text{name}_{1s}$ step, then $\Gamma \vdash e'_i \sim e_m : \tau$.

(2) If $\Gamma \vdash e_i \sim e_m : \tau$ and $e_i \leftrightarrow e'_i$ by a $\beta(1)$, $\beta_{\text{int}}$, look, or $\text{comm}$ step, then $e_m \leftrightarrow e'_m$ and $\Gamma \vdash e'_i \sim e_m : \tau$ for some $e'_m$.

Therefore, if $\Gamma \vdash e_i \sim e_m : \tau$ and $\Gamma \vdash e_i \leftrightarrow e'_i : \tau$ then $e_m \leftrightarrow e'_m$ and $\Gamma \vdash e'_i \sim e_m : \tau$ for some $e'_m$.

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

- $\beta_\rightarrow$ is
  \[ C[\lambda x : \tau.e] a']^\Gamma_\theta(a) \triangleq C[\lambda x : \tau.e]^\Gamma_{\theta}(P[\alpha']^\Gamma_{\theta}(\theta)(a)) \triangleq C[e]^\Gamma_{\{P[a'/x]\theta\}}(a) \triangleq C[e[a'/x]]^\Gamma_\theta(a) \]
  which follows from Lemma 30.

- $\beta_i$ is
  \[ C[\lambda x : \tau.e] \phi]^\Gamma_\theta(a) \triangleq C[\lambda x : \tau.e]^\Gamma_\theta(x)(a) \triangleq C[e]^\Gamma_{\chi}(a) \triangleq C[e]^\Gamma_{\chi}(a) \]
  which follows from Lemma 30.

- $\text{name}_{1s}$ is
  \[ C[I#^\omega e]^\Gamma_\theta(e) \triangleq \text{let } x^\omega_{\text{Intr}} = e \text{ in } I#(x_{\text{Intr}}) \triangleq C[\text{let } x : \text{Int#} = e \text{ in } I#^\omega(x)]^\Gamma_\theta(e). \]

- $\text{name}$ is similar to $\text{name}_{1s}$, and depends on the argument of application and its kind.

The second case of $IL$ reductions that are mirrored by compilation are:

- $\beta(1)$ is
  \[ C[\text{App } (\text{Clos } e) e]^\Gamma_\theta(a) \triangleq \text{App } C[\text{Clos } \theta e]^\Gamma_{\theta}(a) \\rightarrow_{\text{App}} \text{E}_\theta[e]^\Gamma_{\theta}(a) \\rightarrow^2_{\text{Call}} C[e]^\Gamma_\theta(a) \]

assuming $\Gamma : e : \tau$ and $\Gamma : \tau \rightarrow^\text{conv} \eta$, which forces $\overline{\eta} = \text{arity}(\eta)$ and $n = |\overline{\eta}| = |\overline{a}|$ and $\overline{\eta} \sim \overline{a}$.

- $\beta_{\text{int}}$ follows from Lemma 30

\[ C[I#^\omega a'] \text{ of } I# x \rightarrow e[^\Gamma_\theta(a) \triangleq \text{case } I#(P[\alpha']^\Gamma_{\theta})(x_{\text{Intr}}) \rightarrow C[e]^\Gamma_{x\text{Int#}}(a) \\rightarrow_{\text{Unbox}} C[e]^\Gamma_{\{P[a'/x]\theta\}}(a) \triangleq C[e[a'/x]]^\Gamma_\theta(a) \]

- $\text{comm}$ is, for some $\nu$ and $\phi$
  \[ C[F[\text{let } x : \tau = e \text{ in } A]^\Gamma_\theta(a) \leftrightarrow_{\text{Unlift}} C[A^\theta_{\sigma}](F[e]^\Gamma_{\theta}(a)) \\rightarrow_{\text{Alloc}} \text{let } x^\psi_{\text{PrtR}} = \nu \text{ in } C[\text{let } x : \text{int#} = e \text{ in } A]^\Gamma_{\theta}(\epsilon) \]

where $\text{Alloc} = L\text{Alloc}$ if $\psi = L$ and $\text{Alloc} = S\text{Alloc}$ if $\psi = U$. 

\[ \triangleq C[\text{let } x : \tau = e \text{ in } F[A]^\Gamma_\theta(a) \]

\[ (2154, 2149, 2146, 2145, 2144) \]
• look is, by Lemma 27, for some \( v \) and \( \psi 
olimits 
olimits \)

\[
\begin{align*}
C \llbracket x : \tau = e \rrbracket & = e \in E[x] \llbracket x \rrbracket_{\phi}^\Gamma (\alpha) \\
& \overset{U_{\text{Unlift}}}{\leftrightarrow} \llbracket x \rrbracket_{\phi \text{trr}}^\Gamma \llbracket x \rrbracket_{\phi \text{trr}}^\Gamma = E_{\phi} \llbracket e \rrbracket_{\phi}^\Gamma \in C_{\sigma} \llbracket E \rrbracket_{\phi}^\Gamma (\alpha) \llbracket x_{\text{trr}} \rrbracket^\Gamma \\
& \overset{\text{Inline}}{\rightarrow} \llbracket x \rrbracket_{\phi}^\Gamma = E_{\phi} \llbracket e \rrbracket_{\phi}^\Gamma \in C_{\sigma} \llbracket E \rrbracket_{\phi}^\Gamma (\alpha) \llbracket e \rrbracket_{\phi}^\Gamma \\
& \overset{\Delta}{=} C \llbracket x :: x \cdot \tau = e \rrbracket \in E[\llbracket e \rrbracket_{\phi}^\Gamma (\alpha)]
\end{align*}
\]

where Inline = Look or Inline = Memo if \( v = \text{Eval}^Y \) and Inline = Fun if \( v = \text{Call}^[\pi] \).

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

\[
\begin{align*}
C \llbracket E[e] \rrbracket_{\phi}^\Gamma (\alpha) & \overset{\Delta}{=} C_{\tau} \llbracket E \rrbracket_{\phi}^\Gamma (\alpha) \llbracket C \llbracket e \rrbracket_{\phi}^\Gamma (\alpha) \rrbracket \\
& \overset{\tau}{\rightarrow} C_{\tau} \llbracket E \rrbracket_{\phi}^\Gamma (\alpha) \llbracket C \llbracket e \rrbracket_{\phi}^\Gamma (\alpha) \rrbracket \\
& \overset{U_{\text{Unlift}}}{\leftrightarrow} C \llbracket e' \rrbracket_{\phi}^\Gamma (\alpha)
\end{align*}
\]

where additional applications of the Unlift step may be needed when a bound lazy expression (of kind Eval^Y) is finally reduced to a reference \( R \). \( \square \)

Lemma 32 (Backward Simulation). If \( \Gamma \vdash_C e_1 \sim e_m : \tau \) and \( e_m \overset{\tau}{\rightarrow} e'_m \) then \( \Gamma \vdash e_1 \overset{\tau}{\rightarrow} e'_1 \) : \( \tau \) and \( \Gamma \vdash_C e'_1 \sim e'_m \) for some \( e' \).

Proof. Note that if \( \Gamma \vdash_C e_1 \sim e_m : \tau \) then \( e_m \not\vdash \) by the Call or Move steps. Otherwise, given \( e_m \overset{\tau}{\rightarrow} e'_m \) by any other step, it is the case that \( \Gamma \vdash e_1 \overset{\tau}{\rightarrow} e''_1 : \tau \) by \( \beta_{=\ast}, \beta_{\nu}, \text{rename, name}, \) and \( \text{name}_{1,s} \) steps, followed by \( \Gamma \vdash e''_1 \overset{\tau}{\rightarrow} e'_1 : \tau \) by one of the other steps. Therefore, by forward simulation Lemma 25 and determinism of the \( \text{ML} \) operational semantics, we know that \( \Gamma \vdash_C e'_1 \sim e'_m \). \( \square \)

B.5 Correctness of closed compilation

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

(1) Standardization and confluence relating the equational theory of \( \text{IL} \) to the call-by-name operational semantics of \( \text{IL} \) (defined as a sub-relation of the equational theory).
(2) A bisimulation between the call-by-name and call-by-need operational semantics of \( \text{IL} \) based on unwinding (i.e., copying) let-bindings.
(3) A bisimulation between the call-by-need operational semantics of \( \text{IL} \) and the operational semantics of \( \text{ML} \) based on the compilation given in Fig. 6.
(4) A bisimulation between the operational semantics and abstract machine of \( \text{ML} \).

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

Theorem 8 (Soundness and Completeness).

(1) For any \( \vdash e : \text{Int#} \), \( \vdash e = n : \text{Int#} \) if and only if \( \langle \mathcal{E}_{\text{Eval}}^Y [\llbracket e \rrbracket] \mid \llbracket e \rrbracket \rangle \overset{\tau}{\rightarrow} \llbracket n \mid \llbracket e \rrbracket \mid \llbracket H \rrbracket \rangle \).
(2) For any \( \vdash e : \text{Int#} \), \( \vdash e = \# n : \text{Int#} \) if and only if \( \langle \mathcal{E}_{\text{Eval}}^Y [\llbracket e \rrbracket] \mid \llbracket e \rrbracket \rangle \overset{\tau}{\rightarrow} \llbracket \#(n) \mid \llbracket e \rrbracket \mid \llbracket H \rrbracket \rangle \).

Proof. We show the case for \( \text{Int#} \) as \( \text{Int}^Y \) is analogous.

First, from the assumption that \( \vdash e = n : \text{Int#} \), we know:

- \( \vdash e \overset{\tau}{\rightarrow} n : \text{Int#} \) in call-by-name \( \text{IL} \) from Corollary 4,
- \( \vdash e \overset{\tau}{\rightarrow} H[n] : \text{Int#} \) in call-by-need \( \text{IL} \) from Lemmas 12 and 15 and the fact that \( n \) is closed,
- \( \mathcal{E}_{\text{Eval}}^Y [\llbracket e \rrbracket] \overset{\tau}{\rightarrow} H[n] \) in \( \text{ML} \) from Lemmas 28 and 31, and
Types $\sigma$ corresponding to the source call-by-name System F

$$\sigma ::= a \mid \text{Int} \mid \gamma \mid \forall t : \kappa . \sigma \quad \tau ::= \sigma \rightarrow \sigma' \quad \kappa ::= \text{TYPE} \text{PtrR} \text{Eval} \gamma \quad \gamma ::= \text{L}$$

Types $\tau$ corresponding to the source call-by-value System F

$$\tau ::= a \mid \text{Int} \mid \gamma \mid \forall t : \kappa . \sigma \quad \kappa ::= \text{TYPE} \text{PtrR} \text{Eval} \gamma \quad \gamma ::= \text{U}$$

Decompilation of source types $\llbracket \sigma \rrbracket^{-1}$ and target-only types $\llbracket \tau \rrbracket^{-1}$

$\llbracket a \rrbracket^{-1} \triangleq a \quad \llbracket \text{Int} \rrbracket^{-1} \triangleq \text{Int} \quad \llbracket \forall \{ t \} \rrbracket^{-1} \triangleq \llbracket \tau \rrbracket^{-1} \quad \llbracket \forall t : \kappa . \sigma \rrbracket^{-1} \triangleq \forall t . \llbracket \sigma \rrbracket^{-1} \quad \llbracket \sigma \rightarrow \sigma' \rrbracket^{-1}$

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

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

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

Decompilation of serious expressions $\llbracket e \rrbracket^{-1}$ (invariant: $e : \tau$ or $e : \sigma$)

$$\llbracket x \rrbracket^{-1} \triangleq x \quad \llbracket \text{I} \# n \rrbracket^{-1} \triangleq n \quad \llbracket e \ e' \rrbracket^{-1} \triangleq \llbracket e \rrbracket^{-1} \llbracket e' \rrbracket^{-1} \quad \llbracket \lambda x : \sigma . e \rrbracket^{-1} \triangleq \lambda x : \sigma . \llbracket e \rrbracket^{-1} \quad \llbracket \lambda t : \kappa . e \rrbracket^{-1} \triangleq \lambda t . \llbracket e \rrbracket^{-1} \quad \llbracket \text{App} \ e \rrbracket^{-1} \triangleq \llbracket e \rrbracket^{-1} \quad \llbracket \text{Clos} \ y \ V \rrbracket^{-1} \triangleq \eta \llbracket V \rrbracket^{-1}$$

$\eta$-expanded decompilation of values $\eta\llbracket e \rrbracket^{-1}$ (invariant: $e : \tau$ and $e$ is a value)

$$\eta\llbracket \lambda x : \sigma . e \rrbracket^{-1} \triangleq \lambda x : \sigma \llbracket e \rrbracket^{-1} \quad \eta\llbracket \lambda t : \kappa . e \rrbracket^{-1} \triangleq \lambda t . \llbracket e \rrbracket^{-1} \quad \eta\llbracket \text{App} \ e \rrbracket^{-1} \triangleq \eta\llbracket e \rrbracket^{-1} \quad \text{if App} \ e : \sigma \rightarrow \sigma'$$

$$\eta\llbracket \lambda t : \kappa . e \rrbracket^{-1} \triangleq \lambda t . \llbracket e \rrbracket^{-1} \quad \text{if App} \ e : \forall t : \kappa . \sigma$$

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

- $\langle E_{\text{Eval}} \llbracket e \rrbracket \mid e \mid e \rangle \mapsto* \langle n \mid e \mid H \rangle$ from Lemma 25 and Corollary 5.
- Second, from the assumption $\langle E_{\text{Eval}} \llbracket e \rrbracket \mid e \mid e \rangle \mapsto* \langle n \mid e \mid H \rangle$, we know:
  - $E_{\text{Eval}} \llbracket e \rrbracket \mapsto* H[n]$ in $\text{ML}$ from Lemma 26 and Corollary 5,
  - $t \mapsto* H'[n] : \text{Int\#}$ in call-by-need $\text{IL}$ from Lemmas 28 and 32,
  - $t \mapsto* n : \text{Int\#}$ in call-by-name $\text{IL}$ from Lemmas 12 and 13 and the fact that $n$ is closed, and
  - $t \mapsto* n : \text{Int\#}$ from Theorem 5.

\[\square\]

C  CORRECTNESS OF SYSTEM F-TO-$\text{IL}$ COMPILEMENT

Decompilation of types and expressions from $\text{IL}$ 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 $\text{IL}$ 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 $\text{IL}$.

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

1. if $\Gamma \vdash e : \sigma \rightarrow \sigma'$, then $\eta\llbracket e \rrbracket^{-1} = \beta(t) \lambda x : \llbracket \sigma \rrbracket^{-1} . \llbracket e \rrbracket^{-1} x$, and
Kinds are Calling Conventions

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 \(IL\).
2. Given any \( \Gamma \vdash e \) pass in the \(IL\) sublanguage, \([e]\) is a value in System F.
3. Given any \( \Gamma \vdash e \) pass in the \(IL\) sublanguage, \(\eta[e]\) 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 TYPE \(\Pi R_1\) \(Evalu\) so it is a value in \(IL\).

Call-by-value value preservation follows by cases on the forms of values in the source System F and target \(IL\) sublanguage. Note that the \(\eta\)-expansion of \(\eta[App e]\) ensures that this is always a value for types like \(\sigma \sim \sigma'\), from which value preservation follows for \([e]\) for expressions of type \(\sigma\) which all have kind TYPE \(\Pi R_1\) \(Evalu\).

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

1. \(\lbrack \lbrack \tau \rbrack^{-1} \triangleq \tau\), and \(\lbrack \lbrack \sigma \rbrack^{-1} \triangleq \chi\{\sigma\}\).
2. If \(\Gamma \vdash e : \sigma\) then \(\lbrack e \rbrack^{-1} = \beta_{(1, \eta \rightarrow \eta \nu)} e\),
3. If \(\Gamma \vdash e : \tau\) then \(\lbrack[App e]^{-1}\rbrack = \beta_{(1, \eta \rightarrow \eta \nu)} e\), and
4. If \(\Gamma \vdash e : \tau\) and \(\Gamma \vdash e\) pass then \(\lbrack\eta[e]^{-1}\rbrack = \beta_{(1, \eta \rightarrow \eta \nu)} \text{Clos}^\gamma e\).

Proof. By induction on the syntax of \(\tau\) and \(e\), each case following directly from the inductive hypothesis.

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

1. \(\lbrack \lbrack \tau \rbrack^{-1} \triangleq \tau\) and \(\lbrack \lbrack \sigma \rbrack^{-1} \triangleq \chi\{\sigma\}\).
2. If \(\Gamma \vdash e : \sigma\) then \(\lbrack e \rbrack^{-1} = \beta_{(1, \eta \rightarrow \eta \nu)} e\),
3. If \(\Gamma \vdash e : \tau\) then \(\lbrack[App e]^{-1}\rbrack = \beta_{(1, \eta \rightarrow \eta \nu)} e\), and
4. If \(\Gamma \vdash e : \tau\) and \(\Gamma \vdash e\) pass then \(\lbrack\eta[e]^{-1}\rbrack = \beta_{(1, \eta \rightarrow \eta \nu)} \text{Clos}^\gamma e\).

Proof. The first part follows directly on the syntax of \(\tau\) and \(\sigma\), and the remaining parts follow simultaneously by induction on the structure of \(\sigma\). The cases for \(x\) and \(I\#\) \(n\) are immediate, and the remaining cases are as follows:

- (2) \(\lbrack e e' \rbrack^{-1} \triangleq \text{App}[\lbrack e \rbrack^{-1}][\lbrack e' \rbrack^{-1}]:=(3) e[\lbrack e \rbrack^{-1}]==(2) e e'\)
- (2) \(\lbrack e \sigma \rbrack^{-1} \triangleq \chi\{\sigma\}:=(3) e[\lbrack \sigma \rbrack^{-1}]==(2) e[\lbrack \sigma \rbrack^{-1}]=(1) e \sigma\)
- (2) \(\lbrack \lambda t : \kappa . e \rbrack^{-1} \triangleq \lambda t : \kappa . e:\text{Note this case only applies in call-by-name (de)compilation.}\)
- (2) \(\lbrack \text{Clos}^\gamma S \rbrack^{-1} \triangleq \text{Clos}^\gamma S\)
- (3) \(\text{App}[\lbrack e \rbrack^{-1}][\lbrack \sigma \rbrack^{-1}][\lbrack e \rbrack^{-1}]:=(2) e[\lbrack \sigma \rbrack^{-1}]\)+(1) e \sigma\)
- (3) \(\text{App}[\lbrack \lambda t : \kappa . e \rbrack^{-1}]:=(1,2) \beta_{(1, \eta \rightarrow \eta \nu)} \lambda t : \kappa . e\text{ follows analogously to the previous case. Note this case only applies in call-by-value (de)compilation.}\)
- (3) \(\text{App}[\lbrack e \rbrack^{-1}]:=(1,2) \beta_{(1, \eta \rightarrow \eta \nu)} \text{App} e\)
- (4) \(\lbrack \eta[\lambda x : \sigma . e]^{-1}\rbrack \triangleq \lbrack \lambda x : \sigma . e \rbrack^{-1}[\lbrack e \rbrack^{-1}]:=(2) \text{Clos}^\gamma \lambda x : \sigma . e\)


with decompilation (\(\eta\)) analogous to the previous case. Note this case only applies in call-by-value (de)compilation.

(4) Given \(\Gamma \vdash e : \gamma \{\sigma \rightarrow \sigma'\}^\eta\) and \(\eta/\gamma\) call-by-value or call-by-name then \(\eta/\gamma\) follows analogously to the previous case.

**Lemma 37** (Forward Soundness). For any \(\Gamma \vdash e_1 : \tau\) in System F, if \(\Gamma \vdash e_1 = e_2 : \tau\) in either call-by-value or call-by-name then \(\eta/\gamma\) is sound w.r.t. compilation as follows:

- \((\beta_1)\)
  \(\eta/\gamma\) call-by-value or call-by-name then \(\eta/\gamma\) follows analogously to the previous case. \(\square\)

**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:

\[
(\beta_1) \quad ((\lambda x : \tau. e) V) \triangleq \text{App}(\text{Clos}\gamma \lambda x : [\tau]. [e]) [V] =_{\beta_1} \lambda x : [\tau]. [e] [V] =_{\beta_\gamma} e [V/x] \triangleq [e[V/x]]
\]

\[
(\beta_\gamma) \quad ((\lambda t. e) \tau) \triangleq (\lambda t : \kappa. [e]) [\tau] =_{\beta_\gamma} e [\tau/t] \triangleq [e[\tau/t]]
\]

\[
(\eta_1) \quad \lambda x : [\tau](V x) \triangleq \text{Clos}\gamma \lambda x : [\tau]. (\text{App} [V] x) =_{\eta_\gamma} \text{Clos}\gamma (\text{App} [V]) =_{\eta_1} [V]
\]

\[
(\eta_\gamma) \quad \lambda t. V t \triangleq \lambda t. [V] t =_{\eta_\gamma} [V]
\]

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

1. For any \(\Gamma \vdash e_1 : \sigma\), if \(\Gamma \vdash e_1 = e_2 : \sigma\) then \(\eta/\gamma\) call-by-value or call-by-name then \(\eta/\gamma\) is sound w.r.t. compilation as follows:

- \((\beta_\gamma)\)
  \(\eta/\gamma\) call-by-value or call-by-name then \(\eta/\gamma\) follows analogously to the previous case. \(\square\)

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

\[
(\beta_1) \quad ((\lambda x : \tau. e) e')^{-1} \triangleq (\lambda x : [\tau]. [e]^{-1}) [e']^{-1} =_{\beta_1} [e']^{-1} [e[V/x]]^{-1} = [e[e'/x]]^{-1}
\]

\[
(\beta_\gamma) \quad ((\lambda t : \kappa. [e]) [\tau])^{-1} \triangleq (\lambda t : [\kappa]. [e])^{-1} [\tau]^{-1} =_{\beta_\gamma} [e][\tau/t]^{-1}
\]
• \((\beta_i)\), Given that \(\Gamma \vdash e : \sigma \leadsto \sigma'\), we have

\[
\eta[\text{App } (\text{Clos}^Y e)]^{-1} \triangleq \lambda x : [\sigma]^{-1}.[\text{Clos}^Y e]^{-1} x \triangleq \lambda x : [\sigma]^{-1}.\eta[e]^{-1} x = _{\eta(1)} \eta[e]^{-1}
\]

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

• \((\eta_\sim)\)

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

• \((\eta_\nu)\) In call-by-name, we have

\[
[\lambda t : \kappa.(e t)]^{-1} \triangleq \lambda t.([\eta[e]^{-1} t) = _{\eta_\nu} [\eta[e]^{-1}
\]

whereas in call-by-value, we have an analogous equality to the previous case.

• \((\eta_1)\) Given that \(\Gamma \vdash V : \forall \gamma \sigma \leadsto \sigma'\)

\[
[\text{Clos}^Y (\text{App } e)]^{-1} \triangleq \eta[\text{App } e]^{-1} \triangleq \lambda x : [\sigma]^{-1}.(\eta[e]^{-1} x) = _{\eta(1)} [V]^{-1}
\]

The case in call-by-value where \(\Gamma \vdash e : \forall \gamma \sigma\leadsto \sigma'\) 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 \(\text{App } e\) expansion of type \(\sigma \leadsto \sigma'\), which simplify to a known case as follows:

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

\[
\eta[\text{App } e]^{-1} \triangleq \lambda x : [\sigma]^{-1}.(\eta[e]^{-1} x) = _{\beta_i} \eta[e]^{-1}
\]

An analogous derivation is also required for polymorphic application and \(\text{App } e\) expansion in call-by-value, which both follow from the \(\beta_\nu\) axiom. \(\square\)

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

**Corollary 7** (Soundness and Completeness). For any \(\Gamma \vdash e_1 : \tau\) in either call-by-name or call-by-value System F, \(\Gamma \vdash e_1 = e_2 : \tau\) if and only if \([\Gamma] [\forall \tau e_1] = [\forall \tau e_2] = [\tau]^{-1}\).