Levity Polymorphism (extended version)

Richard A. Eisenberg
Bryn Mawr College
Bryn Mawr, PA, USA
rae@cs.brynmawr.edu

Simon Peyton Jones
Microsoft Research
Cambridge, UK
simonpj@microsoft.com

Abstract

Parametric polymorphism is one of the lynchpins of modern typed programming. A function that can work seamlessly over a variety of types simplifies code, helps to avoid errors introduced through duplication, and is easy to maintain. However, polymorphism comes at a very real cost, one that each language with support for polymorphism has paid in different ways. This paper describes this cost, proposes a theoretically simple way to reason about the cost—that kinds, not types, are calling conventions—and details one approach to dealing with polymorphism that works in the context of a language, Haskell, that prizes both efficiency and a principled type system.

This approach, levity polymorphism, allows the user to abstract over calling conventions; we detail and verify restrictions that are necessary in order to compile levity-polymorphic functions. Levity polymorphism has opened up surprising new opportunities for library design in Haskell.

1. The cost of polymorphism

Consider the following Haskell function:

\[
bTwice :: \forall a. \text{Bool} \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a
\]
\[
bTwice \; b \; x \; f = \text{case}\; b\;\text{of}\;\text{True} \rightarrow f\; (f\; x)\;\text{False} \rightarrow x
\]

The function is polymorphic in \( a \); that is, the same function works regardless of the type of \( x \), provided \( f \) and \( x \) are compatible. When we say “the same function” we usually mean “the same compiled code for \( bTwice \) works for any type of argument \( x \)”. But the type of \( x \) influences the calling convention, and hence the executable code for \( bTwice \)!

For example, if \( x \) were a list, it would be passed in a register pointing into the heap; if it were a double-precision float, it would be passed in a special floating-point register; and so on. Thus, sharing code conflicts with polymorphism.

A simple and widely-used solution is this: represent every value uniformly, as a pointer to a heap-allocated object. That solves the problem, but it is terribly slow (Section 2.1). Thus motivated, most polymorphic languages also support some form of unboxed values that are represented not by a pointer but by the value itself.

1 We use the term polymorphism to refer exclusively to parametric polymorphism.

context is the Glasgow Haskell Compiler (GHC), a state of the art optimizing compiler for Haskell. It has had unboxed values for decades, but not without the inevitable tensions that arise between unboxed values and polymorphism (Section 3). Other languages deal differently with this challenge (Section 8).

In this paper we describe an elegant new approach to reconciling high performance with pervasive polymorphism. Our contributions are as follows:

- We present (Section 4) a principled way to reason about compiling polymorphic functions and datatypes, by categorizing types into kinds. Each kind describes the memory layout of its types, thus determining the calling convention of functions over those types.
- Having a principled way to describe memory layout and calling convention, we go one step further and embrace levity polymorphism, allowing functions to be abstracted over choices of memory layout provided that they never move or store data with an abstract representation (Section 5). We believe we are the first to describe and implement levity polymorphism.
- It is tricky to be sure precisely when it is, and is not, OK to permit levity polymorphism. We give a formal proof that our rules are sufficient to guarantee that levity-polymorphic functions can indeed be compiled into concrete code (Section 6).
- With levity polymorphism in hand, a range of new possibilities open up—including the ability to write an informative kind for \((-\to\)\) and to overload operations over both boxed and unboxed types in a principled way. We explore these in Section 7.

Levity polymorphism is implemented in GHC, version 8.0.1, released early 2016. We are not the first to use kinds in this way—Cyclone uses a similar approach Section 8.1—but we take the idea much further than any other compiler we know, with happy consequences. Remember: it’s all about performance. If you don’t care about performance, life is much simpler!

2. Background: performance through unboxed types

We begin by describing the performance challenges that our paper tackles. We use the language Haskell and the compiler GHC as a concrete setting for this discussion, but many of our observations apply equally to other languages supporting polymorphism. We discuss other languages and compilers in Section 8.

2.1 Unboxed values

Consider this loop, which computes the sum of the integers 1 \ldots n:

\[
\sum_{i=1}^{n} i
\]

\[2\text{ GHC extends Haskell in many ways to better support high-performance code, so when we say “Haskell” we will always mean “GHC Haskell”}\]
Figure 1. Boxity and levity, with examples

<table>
<thead>
<tr>
<th>Lifted</th>
<th>Unboxed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Int, Char, ByteArray</td>
</tr>
</tbody>
</table>

\[
\text{sumTo} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{sumTo} \ acc \ 0 = acc \\
\text{sumTo} \ acc \ n = \text{sumTo} \ (acc + n) \ (n - 1)
\]

GHC represents a values of type \text{Int} as a pointer to a two-word heap-allocated cell; the first word is a descriptor, while the second has the actual value of the \text{Int}. If \text{sumTo} used this representation throughout, it would be unacceptably slow. Each iteration would evaluate its second argument \text{acc} follow the the pointer to get the value, and test it against zero; in the non-zero case it would allocate thunks for \text{acc} and \text{acc} + \text{n}, and then iterate. In contrast, a C compiler would use a three-machine-instruction loop, with no memory traffic whatsoever. The performance difference is enormous.

GHC therefore provides a built-in data type \text{Int}

\[
\text{Int}
\]

\[
\text{Int}
\]

\[
\text{Int}
\]

\[
\text{Int}
\]

\[
\text{Int}
\]

\[
\text{Int}
\]

\[
\text{Int}
\]

An \text{Int} is represented by a pointer into the heap, so performance-conscious programmers often program with \text{Int} and its algebraic operations, and boxes the result \text{sumTo} \ (acc + n) \ (n - 1) and the pointer to get the value, and test it against zero; in the non-zero case it would allocate thunks for \text{acc} and \text{acc} + \text{n}, and then iterate. In contrast, a C compiler would use a three-machine-instruction loop, with no memory traffic whatsoever. The performance difference is enormous.

GHC therefore provides a built-in data type \text{Int} of unboxed integers. An \text{Int} is represented by a pointer but by the integer itself. Now we can rewrite \text{sumTo} like this:

\[
\text{sumTo} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{sumTo} \ acc \ 0 = acc \\
\text{sumTo} \ acc \ n = \text{sumTo} \ (acc + n) \ (n - 1)
\]

We had to use different arithmetic operations and literals, but apart from that the source code looks just the same. But the compiled code is very different; we get essentially the same code as if we had written it in C.

GHC’s strictness analyzer and other optimizations can often transform \text{sumTo} into \text{sumTo} \ _\text{pure}. But it cannot guarantee to do so, so performance-conscious programmers often program with unboxed values directly. As well as \text{Int}, GHC provides a solid complement of other unboxed types, such as \text{Char} and \text{Char}, together with primitive operations that operate on them. Given these unboxed values, the boxed versions can be defined in Haskell itself; GHC does not treat them specially. For example:

\[
\text{data Int} = l_p \ \\
\text{plusInt} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{plusInt} \ (l_p \ i_1) \ (l_p \ i_2) = l_p \ (i_1 + i_2)
\]

Here Int is an ordinary algebraic data type, with one data constructor \text{l_p}, that has one field of type \text{Int}. The function \text{plusInt} simply pattern matches on its arguments, fetches their contents \text{i_1} and \text{i_2}, both of type \text{Int}, adds them using \text{(+)} \text{acc} and \text{acc} + \text{n}, and boxes the result with \text{l_p}.

2.2 Boxed vs. unboxed and lifted vs. unlifted

In general, a boxed value is represented by a pointer into the heap, while an unboxed value is represented by the value itself. It follows that an unboxed value cannot be a thunk; arguments of unboxed type must be passed by value.

Haskell also requires consideration of levity—that is, the choice between lifted and unlifted. A lifted type is one that is lazy. It is considered lifted because it has one extra element beyond the usual values, representing a non-terminating computation. For example, Haskell’s \text{Bool} type is lifted, meaning that three \text{Bools} are possible: \text{True}, \text{False}, and \text{⊥}. An unlifted type, on the other hand, is strict. The element \text{⊥} does not exist in an unlifted type.

Because Haskell represents lazy computation as thunks at runtime, all lifted types must also be boxed. However, it is possible to have boxed, unlifted types. Figure 1 summarizes the relationship between boxity and levity, providing examples of the three possible points in the space.

2.3 Unboxed tuples

Along with the unboxed primitive types (such as \text{Int} and \text{Double}), Haskell has support for unboxed tuples. A normal, boxed tuple—of type, say, \text{(Int, Bool)}—is represented by a heap-allocated vector of pointers to the elements of the tuple. Accordingly, all elements of a boxed tuple must also be boxed. Boxed tuples are also lazy, although this aspect of the design is a free choice.

Originally conceived to support returning multiple values from a function, an unboxed tuple is merely Haskell syntax for tying multiple values together. Unboxed tuples do not exist at runtime, at all. For example, we might have

\[
\text{divMod} :: \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int}, \text{Int})
\]

that returns two integers. A Haskell programmer might use \text{divMod} like this,

\[
\text{case divMod} \ n \ k \ of \ (\text{_, rem}) \rightarrow ...
\]

using \text{case} to unpack the components of a tuple. However, during compilation, the unboxed tuple is erased completely. The \text{divMod} function is compiled to return two values, in separate registers, and the \text{case} statement is compiled simply to bind \text{quot} and \text{rem} to those two values. This is more efficient than an equivalent version with boxed tuples, avoiding allocation for the tuple and indirection.

Modern versions of GHC also allow unboxed tuples to be used as function arguments: \text{(+)} :: \text{(Int, Int) → Int} compiles to the exact same code as \text{(+)} :: \text{Int → Int} → \text{Int}; the unboxed tuple is used simply to represent multiple arguments passed via multiple registers.

An interesting aspect of unboxed tuples, important to the story in this paper, is that nesting is computationally irrelevant. That is, while \text{(Int, (Float, Bool))} is a distinct type from \text{(Int, Float, Bool)}, the two are identical at runtime; both represent three values to be passed or returned via three registers.

3. Unboxed types and polymorphism

Recall the function \text{bTwice} from the introduction:

\[
\text{bTwice} :: \forall a. \text{Bool} \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a
\]

\[
\text{bTwice} \ b \times f = \text{case b of True} \rightarrow f \ (f \ x)
\]

\[
\text{False} → x
\]

Like many other compilers for a polymorphic language, GHC assumes that a value of polymorphic type, such as \text{x :: a}, is represented uniformly by a heap pointer. So we cannot call \text{bTwice} with \text{x :: Int} \text{or x :: Float}, or indeed with \text{x :: (Int, Int)}. Actually, \text{bTwice} cannot even be used on a boxed unlifted value, such as a \text{ByteArray}. Why not? Because if \text{a} is unlifted the call \text{(f (f \ x))} should be compiled using call-by-value, whereas if \text{a} is a lifted type the call should to be compiled with call-by-need.

GHC therefore adopts the following principle:

- \text{The Instantiation Principle}. You cannot instantiate a polymorphic type variable with an unlifted type.

That is tiresome for programmers, but in return they get solid performance guarantees. (An alternative would be some kind of
auto-specialization, as we discuss in Section[8]. However, adopting the instantiation principle turns out to be much less straightforward than it sounds, as we elaborate in the rest of this section. These are the challenges that we solve in the rest of the paper.

3.1 Kinds

How can the compiler implement the instantiation principle? For example, how does it even know if a type is unlifted?

Haskell classifies type by kinds, much the same way that terms are classified by types. For example, $\text{Bool} :: \text{Type}$, $\text{Maybe} \text{Bool} :: \text{Type}$. So it is natural to use the kind to classify types into the lifted and unlifted forms, thus $\text{Int}_{\#} :: \#$, $\text{Float}_{\#} :: \#$, where “$\#$” is a new kind that classifies unlifted types. \[6\]

In contrast, $\text{Type}$ classifies lifted types and, because of laziness, a value of lifted type must be represented uniformly by a pointer into the heap. So the Instantiation Principle can be refined to this: all polymorphic type variables have kind $\text{Type}$. For example, here is $\text{Bitwice}$ with an explicitly-specified kind:

$$b\text{Twice} :: \forall (a :: \text{Type}). \text{Bool} \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a$$

Now we attempt to instantiate it at type $\text{Float}_{\#} :: \#$; we will get a kind error because $\text{Type}$ and $\#$ are different kinds.

3.2 Sub-kinding

Haskell has rich language of types. Of particular interest is that the function arrow: $\rightarrow$ is just a binary type constructor with kind $$(\rightarrow) :: \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$$

Partial applications of $(\rightarrow)$ are often useful:

$$\text{instance Monad} \ ((\rightarrow) \text{ env}) \text{ where }$$

$$\text{return } x = \lambda \text{env} \rightarrow x$$

$$\text{ma } >>= \text{fmb} = \lambda \text{env} \rightarrow \text{fmb} (\text{ma env} \text{ env})$$

Given the above kind for $(\rightarrow)$, the type $\text{Monad} \ ((\rightarrow) \text{ env})$ is well-kindred. We have $(\rightarrow) \text{ env} :: \text{Type} \rightarrow \text{Type}$, and that is the kind that $\text{Monad}$ expects.

But now we have a serious problem: a function over unified types, such as $\text{sumTo} :: \text{Int}_{\#} \rightarrow \text{Int}_{\#} \rightarrow \text{Int}_{\#}$, becomes ill-kindred! Why? Because $(\rightarrow)$ expects a $\text{Type}$, but $\text{Int}_{\#} :: \#$. This problem has dogged GHC ever since the introduction of unboxed values. For many years its “solution” was to support a sub-kinding relation, depicted here:

```
<table>
<thead>
<tr>
<th>OpenKind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>#</td>
</tr>
</tbody>
</table>
```

That is, GHC had a kind $\text{OpenKind}$, a super-kind of both $\text{Type}$ and $\#$. We could then say that $(\rightarrow) :: \text{OpenKind} \rightarrow \text{OpenKind} \rightarrow \text{Type}$

To avoid the inevitable complications of sub-kinding and kind inference, GHC also stipulated that only fully-saturated uses of $(\rightarrow)$ would have this bizarre kind; partially applied uses of $(\rightarrow)$ would get the far saner kind $\text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$ as we have seen above.

Haskellers paid for this sleight-of-hand, of course:

- Keen students of type theory would, with regularity, crop up on the mailing lists and wonder why, when we can see that $(\rightarrow) :: \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$, GHC accepts types like $\text{Int}_{\#} \rightarrow \text{Double}_{\#}$.
- It is well known that the combination of (a) type inference, (b) polymorphism, and (c) sub-typing, is problematic. And indeed GHC’s implementation of type inference was riddled with awkward and unprincipled special cases caused by sub-kinding.
- The introduction of kind polymorphism[17] made this situation worse, and the subsequent introduction of kind equalities[16] made it untenable.
- The kind $\text{OpenKind}$ would embarrassingly appear in error messages.

All in all, the sub-kinding solution was never satisfactory and was screaming to us to find something better.

3.3 Functions that diverge

Consider this function

$$f :: \text{Int}_{\#} \rightarrow \text{Int}_{\#}$$

$$f \ n = \text{if } n <_{\#} 0_{\#} \text{ then error "Negative argument" then else } n \# 2_{\#}$$

Here error ::\forall a. String $\rightarrow$ a prints the string and halts execution[7]

But under the Instantiation Principle, this call to error should be rejected, because we are instantiating $a$ with $\text{Int}_{\#}$. But in this case, it is OK to break the Instantiation Principle! Why? Because error never manipulates any values of type $a$—it simply halts execution. It is tiresome for a legitimate use of error to be rejected in this way, so GHC has given error a magical type

$$\forall (a :: \text{OpenKind}). \text{String} \rightarrow a$$

Now, using the sub-kinding mechanism described above, the call can be accepted. Alas, the magic is fragile. If the user writes a variant of error like this:

$$\text{myError} :: \text{String} \rightarrow a$$
$$\text{myError} \ s = \text{error "Program error " } + \ s$$

then GHC infers the type $\forall (a :: \text{Type}). \text{String} \rightarrow a$, and the magic is lost.

4. Key idea: polymorphism, not sub-kinding

We can now present the main idea of the paper: replace sub-kinding with kind polymorphism. As we shall see, this simple idea not only deals neatly with the awkward difficulties outlined above, but it also opens up new and unexpected opportunities (Section[7]). Using polymorphism as a replacement for a sub-typing system is not a new idea; for example see Finne et al. [2], where Section 5 is entitled “Polymorphism expresses single inheritance”. However, even starting down this road required the rich kinds that have only recently been added to GHC [16][17]; this new approach was not properly conceivable earlier.

4.1 Runtime-representation polymorphism

Here is the design, as implemented in GHC 8[8]. We introduce a new, primitive type-level constant, $\text{TYPE}$.

\[7\] More precisely, it throws an exception

\[8\] The design presented here is actually evolved in a few ways from what has been released. For example, GHC 8 defines $\text{Rep}$ as the enumeration instead of a list of unary representations.
TYPE :: Rep → Type

with the following supporting definitions.

\[
\begin{align*}
\text{type } & \text{Rep} = \text{UnaryRep} & \text{-- A type-level list} \\
\text{data } & \text{UnaryRep} = \text{PtrRep} & \text{-- Boxed, lifted} \\
& \text{UPtrRep} & \text{-- Boxed, unlifted} \\
& \text{IntRep} & \text{-- Unboxed ints} \\
& \text{FloatRep} & \text{-- Unboxed floats} \\
& \text{DoubleRep} & \text{-- Unboxed doubles} \\
& \ldots \text{ etc} \\
\text{type } & \text{Lifted} = \text{[PtrRep]} \\
\text{type } & \text{Type} = \text{TYPE Lifted}
\end{align*}
\]

Rep is a type that describes the runtime representation of values of a type. Type, the kind that classifies the types of values, was previously treated as primitive, but now becomes a synonym for TYPE Lifted, where Lifted :: Rep. It is easiest to see how these definitions work using examples:

\[
\begin{align*}
\text{Int} & :: \text{Type} \\
\text{Int} & :: \text{TYPE Lifted} & \text{-- Expanding Type} \\
\text{Int} & :: \text{TYPE } \text{[PtrRep]} & \text{-- Expanding Lifted} \\
\text{Int} & :: \text{TYPE } \text{[IntRep]} \\
\text{Float} & :: \text{TYPE } \text{[FloatRep]} \\
\text{Int} & :: \text{TYPE } \text{[Int]} \\
\text{Maybe Int} & :: \text{Type} \\
\text{Maybe} & :: \text{Type } \rightarrow \text{Type}
\end{align*}
\]

Any type that classifies values, whether boxed or unboxed, lifted or unlifted, has kind TYPE r for some r :: Rep. The type Rep specifies how a value of that type is represented, by giving a list of UnaryRep. Typically this list has exactly one element; we will see why a list is useful in Section 4.2.

A UnaryRep specifies how a single value is represented, where by “a single value” we mean one that can be stored in a typical machine register and manipulated by a single machine instruction. Such a values include: a heap pointer to a lifted value (PtrRep); a heap pointer to an unlifted value (UPtrRep); an unboxed fixed-precision integer value (IntRep); an unboxed floating-point value (FloatRep), and so on. Where we have multiple possible precisions we have multiple constructors in UnaryRep; for example we have DoubleRep as well as FloatRep.

The type UnaryRep is not magic: it is a perfectly ordinary algebraic data type, promoted to the kind level by GHC’s DataKinds extension [10]. Similarly, Rep, Lifted, and Type are all perfectly ordinary type synonyms. Only TYPE is primitive in this design. It is from these definitions that we claim that a kind dictates a type’s representation, and hence its calling convention. For example, Int and Bool have the same kind, and hence use the same calling convention. But Int# belongs to a different kind, and uses a different calling convention.

There are, naturally, several subtleties, addressed in the subsections below.

4.2 Representing unboxed tuples

Why did we define Rep as a list of UnaryRep? It is so that we can represent the kinds of unboxed tuples:

\[
\begin{align*}
\text{(Int, Bool)} & :: \text{TYPE } \text{[PtrRep, PtrRep]} \\
\text{(Int#), (Bool#)} & :: \text{TYPE } \text{[IntRep, PtrRep]} \\
\text{(Int, #, Bool)} & :: \text{TYPE } \text{[]} \\
\end{align*}
\]

In GHC Haskell, you can use lists at the type level, with a tick-mark to identify a type-level list. Here [UnaryRep] is the list of lists containing elements of type UnaryRep, while [PtrRep] is a list containing one element, PtrRep.

An unboxed pair is represented by two registers, an unboxed triple by three, and so on. An unboxed empty tuple is represented by no registers, a surprisingly useful value in practice.

Note that this format respects the computational irrelevance of nesting of unboxed tuples. For example these three types all have the same kind, here written PFP for short:

\[
\begin{align*}
\text{type PFP = TYPE } & \text{[PtrRep, FloatRep, PtrRep]} \\
& \text{(Int, Float, Bool)} :: \text{PFP} \\
& \text{(Int, Float, Bool)} :: \text{PFP} \\
& \text{(Int, Float, Bool)} :: \text{PFP}
\end{align*}
\]

A value of any of these three types is represented by two pointers to boxed lifted values, and an unboxed float. That does not that the three types are equal or interchangeable — after all, Int :: Type and Bool :: Type but that does not mean that Int and Bool are equal! — but all three are represented in the same way, and share the same calling convention.

4.3 Levity polymorphism

We can now give proper types to (→) and error:

\[
\begin{align*}
\text{(→) :: } & \forall (r1 :: \text{Rep}) (r2 :: \text{Rep}) \\
& \text{TYPE } r1 \rightarrow \text{TYPE } r2 \rightarrow \text{Type} \\
\text{error :: } & \forall (r :: \text{Rep}) (a :: \text{TYPE } r). \text{String } \rightarrow a
\end{align*}
\]

These types are polymorphic in r :: Rep. We call such abstraction “levity polymorphism” — a name owing to its birth as an abstraction over only the levity (lifted vs. unlifted) of a type. It might now properly be called representation polymorphism, but we prefer the original terminology as briefer and more recognizable — that is, easier to search for on a search engine.

Levity polymorphism adds new, and useful, expressiveness to the language (Section 7), but it needs careful handling as we discuss in Section 5.

4.4 The kind of TYPE

Above, we gave the type of TYPE as Rep → Type. That look suspicious because Type is short for TYPE Lifted, so the kind of TYPE involves TYPE. Is that OK?

Yes it is. Unlike other dependently typed languages, GHC does not stratify the universes of types, and instead supports the axiom Type :: Type [16]. While this choice of design makes the language inconsistent when viewed as a logic, it does not imperil type safety. The type safety point is addressed in other work [11][16]; we do not revisit it here.

You might also wonder whether why TYPE returns a TYPE Lifted. Why not return TYPE \[\text{[IntRep]}\], or one of the other possibilities? What does it even mean to talk of the representation of a type?

We choose TYPE :: Rep → Type because it supports a future extension to a full-spectrum dependently-typed language in which types are first-class values and can be passed at runtime. What would it mean to pass a type at runtime? Presumably it would mean passing a pointer to a heap-allocated syntax tree describing the type; so Type would be the appropriate return kind.

5. Taming levity polymorphism

In its full glory, levity polymorphism is un-compilable, at least not without runtime code generation. Let us return to our initial example of bTwice. Would this work?

\[
bTwice :: \forall (r :: \text{Rep}) (a :: \text{TYPE } r). \\
\text{Bool } \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a
\]

Sadly, no. We cannot compile a levity-polymorphic bTwice into concrete machine code, because its calling convention depends on r.
One possibility is to generate specialized versions of bTwice, perhaps at runtime. That choice comes with significant engineering challenges, albeit less so in a JIT-based system. (See Section 5.5 for one setting where this has been done.) Here we explore the alternative: how to restrict the use of levity polymorphism so that it can be compiled.

5.1 Rejecting un-compilable levity polymorphism

The fundamental requirement is this:

Never move or store a levity-polymorphic value. (*)

Note that it is perfectly acceptable for a machine to store a value of a polymorphic type, as long as it is not evaluated. The second argument is passed in as a pointer, and the result is returned as one. There is no need to know a concrete type of the data these pointers refer to. Yet we do need to know the kind of these types, to fix the calling conventions of the arguments and return value.

We now turn our attention to ensuring that (*) holds, a property we attain via two restrictions:

1. Disallow levy-polymorphic binders. Every bound term variable in a Haskell program must have a type whose kind is fixed and free of any type variables. This rule would be violated had we implemented bTwice with the type as given in this section: we would have to bind a variable of type `a :: TYPE r`.

2. Disallow levy-polymorphic function arguments. Arguments are passed to functions in registers. During compilation, we need to know what size register to use.

These checks can be easily performed after type inference is complete. Any program that violates these conditions is rejected. We prove that these checks are sufficient to allow compilation in Section 6.

5.2 Type inference and levy polymorphism

Phrasing the choice between the concrete instantiations of `TYPE` as the choice of a `Rep` is a boon for GHC’s type inference mechanism. When GHC is checking an expression (`λx → e`), it must decide on a type for `x`. The algorithm naturally invents a unification variable `ρ`. But what kind does `α` have? Equipped with levy polymorphism, GHC invents another unification variable `ρ :: Rep` and chooses `α :: TYPE ρ`. If `x` is used in a context expecting a lifted type, then `ρ` is unified with `Lifted`—all using GHC’s existing unification machinery. In terms of GHC’s implementation, this is actually a simplification over the previous sub-kinding story.

However, we must be careful to enforce the restrictions of Section 5.1. For example, consider defining `f x = x`. What type should we infer for `f`? If we simply generalized over unconstrained unification variables in the usual way, we would get

\[ f :: \forall (r :: Rep) (a :: TYPE r), a \rightarrow a \]

but, as we have seen, that is un-compilable because its calling convention depends on `r`. We could certainly track all the places where the restrictions of Section 5.1 apply; but that is almost everywhere, and results in a great deal of busy-work in the type inference engine. So instead we never infer levy polymorphism, but we can for the first time check the declared uses of levy polymorphism. Thus, we can write

\[ myError :: \forall (r :: Rep) (a :: TYPE r). String \rightarrow a myError s = error "Program error " + s \]

to get a levy-polymorphic `myError`. Alternatively, we can omit the signature in which case GHC will infer a levy-monomorphic type thus: any levy variable that in principle could be generalized is instead defaulted to `Type`.

Finally, any attempt to declare the above levy-polymorphic type signature for `f` will fail the check described in Section 5.1.

6. Correctness of levy polymorphism

We claim above (Section 5.1) that restricting the use of levy polymorphism in just two ways means that we can always compile programs to concrete machine code. Here, we support this claim by proving that a levy-polymorphic language with exactly these restrictions is compilable.

First, we define `L`, a variant of System F [4][14] that supports levy polymorphism. Second, we define a lower-level language `M`, a λ-calculus in A-normal form (ANF) [3]. Its operational semantics works with an explicit stack and heap and is quite close to how a concrete machine would behave. All operations must work with data of known, fixed width; `M` does not support levy polymorphism. Lastly, we define type-erasing compilation as a partial function from `L` to `M`. We prove our compilation function correct via two theorems: that compilation is well-defined whenever the source `L`-expression is well-typed and that the `M` operational semantics simulates that for `L`.

6.1 The `L` language

The grammar for `L` appears in Figure 2. Along with the usual System F constructs, it supports the base type `Int`, with literals `n`; data constructor `lp` to form `Int`; case expressions for unboxing integers; and `error`. Most importantly, `L` supports levy polymorphism via the novel forms `Λr. e` and `e ρ`, abstractions over and applications to

\[ x \quad Variables \quad α \quad Type variables \quad n \quad Integer literals \quad r \quad Representation variables \]

| v | ::= | P | l |
| ρ | ::= | r | v |
| κ | ::= | TYPE ρ |
| B | ::= | Int | Int # |
| τ | ::= | B | τ₁ → τ₂ | α | ∀α. τ | rτ |
| e | ::= | x | e₁ e₂ | λx:τ. e | λκ. e | e τ | Λ r. e | e ρ | lp | e |
| | | | case e₁ of lp[x] → e₂ | n | error |
| v | ::= | λκ. e | λκ. v | Λ r. v | lp[v] | n |

Values

| Γ | ::= | ∅ | Γ, x:τ | Γ, κ:α | Γ, r |

Types

| τ | ::= | x |

Base types

| ρ | ::= | x |

Kinds

| κ | ::= | |∀ |

Figure 2. The grammar for `L`

12 Refusing to generalize over type variables of kind `Rep` is quite like Haskell’s existing `monomorphic restriction`, where certain unconstrained type variables similarly remain ungeneralized. Both approaches imperil having principal types. In the case of levy polymorphism, the most general type for `f` is un-compilable, so the loss of principal types is inevitable. However, all is not lost: a program that uses boxed types (as most programs do) retains principal types within that fragment of the language.

13 As we explore in Section 6.4 there is one lemma in this proof that we assume the correctness of. This lemma would be necessary for any proof that a compilation to ANF is sound in a lazy language and is not at all unique to our use of levy polymorphism.
The main payload of \( \Gamma \vdash \alpha : \kappa \): the kind \( \kappa \) of \( \alpha \). This kind \( \kappa \) is not part of \( \alpha \)’s type \( \tau \); instead, \( \kappa \) is a type parameter in \( \alpha \)’s parametricity constraint \( \kappa \). We must ensure that \( \kappa \) is not erased.

The main payload of \( \Gamma \vdash \alpha : \kappa \): the kind \( \kappa \) of \( \alpha \). This kind \( \kappa \) is not part of \( \alpha \)’s type \( \tau \); instead, \( \kappa \) is a type parameter in \( \alpha \)’s parametricity constraint \( \kappa \). We must ensure that \( \kappa \) is not erased.

The main payload of \( \Gamma \vdash \alpha : \kappa \): the kind \( \kappa \) of \( \alpha \). This kind \( \kappa \) is not part of \( \alpha \)’s type \( \tau \); instead, \( \kappa \) is a type parameter in \( \alpha \)’s parametricity constraint \( \kappa \). We must ensure that \( \kappa \) is not erased.

The main payload of \( \Gamma \vdash \alpha : \kappa \): the kind \( \kappa \) of \( \alpha \). This kind \( \kappa \) is not part of \( \alpha \)’s type \( \tau \); instead, \( \kappa \) is a type parameter in \( \alpha \)’s parametricity constraint \( \kappa \). We must ensure that \( \kappa \) is not erased.

The main payload of \( \Gamma \vdash \alpha : \kappa \): the kind \( \kappa \) of \( \alpha \). This kind \( \kappa \) is not part of \( \alpha \)’s type \( \tau \); instead, \( \kappa \) is a type parameter in \( \alpha \)’s parametricity constraint \( \kappa \). We must ensure that \( \kappa \) is not erased.

The main payload of \( \Gamma \vdash \alpha : \kappa \): the kind \( \kappa \) of \( \alpha \). This kind \( \kappa \) is not part of \( \alpha \)’s type \( \tau \); instead, \( \kappa \) is a type parameter in \( \alpha \)’s parametricity constraint \( \kappa \). We must ensure that \( \kappa \) is not erased.
Metavariables:

\[ p \] Lifted (pointer) variables

\[ i \] Integer variables

\[ y ::= p | i \]

Variables

\[ t ::= t y | t n | \lambda y . t | y \leftarrow p = t_i \text{ in } t_2 \]

\[ \text{let}! y = t_i \text{ in } t_2 \]

\[ \text{case } t_1 \text{ of } I_{y}[y] \rightarrow t_2 | \text{error} \]

Expressions

\[ w ::= \lambda y . t \mid t_i[n] \mid n \]

Stacks

\[ S ::= \emptyset \mid \text{Force}(p), S \mid \text{App}(p), S \mid \text{App}(n), S \]

Heaps

\[ H ::= \emptyset \mid p \rightarrow t , H \]

Machine states

\[ V ::= \emptyset \mid x \rightarrow y, V \mid y, V \]

Variable envs.

---

**Figure 5.** The grammar for \( \mathcal{M} \)

\[
\begin{align*}
(t p; S; H) & \rightarrow (t; \text{App}(p), S; H) \quad \text{PAPP} \\
(t n; S; H) & \rightarrow (t; \text{App}(n), S; H) \quad \text{IAPP} \\
(p; S; p \rightarrow w, H) & \rightarrow (w; S; p \rightarrow w, H) \quad \text{VAL} \\
(p; S; p \rightarrow t, H) & \rightarrow (t; \text{Force}(p), S; H) \quad \text{EVAL} \\
\{\text{let} \rightarrow t_i \text{ in } t_2; S; H\} & \rightarrow (t_2; S \rightarrow \text{Let}(t_i, S, H)) \quad \text{LET} \\
\{\text{let}! y = t_i \text{ in } t_2; S; H\} & \rightarrow (t_2; S \rightarrow \text{Case}(y, t_2, S, H)) \quad \text{SLET} \\
\{\text{case } t_1 \text{ of } I_{y}[y] \rightarrow t_2; S; H\} & \rightarrow (t_2; S \rightarrow \text{Case}(t_1, y, t_2, S, H)) \quad \text{CASE} \\
\{\text{error} \rightarrow \_\} & \rightarrow \_ \quad \text{ERR} \\
(\lambda p_1 . p_2; S; H) & \rightarrow (t_i [p_1/p_2]; S; H) \quad \text{PPPOP} \\
(\lambda p_1 . p_2; S; H) & \rightarrow (t_i [p_1/i]; S; H) \quad \text{IPPOP} \\
\langle w; \text{Force}(p), S; H\rangle & \rightarrow (w; S; p \rightarrow w, H) \quad \text{FCE} \\
\langle n; \text{Let}(i, t), S; H\rangle & \rightarrow (t [n/i]; S; H) \quad \text{ILET} \\
\{I_{y}[n]; \text{Case}(i, t), S; H\} & \rightarrow (t [n/i]; S; H) \quad \text{IMAT}
\end{align*}
\]

**Figure 6.** Operational semantics for \( \mathcal{M} \)

---

only on variables or literals. We accordingly need to be able to let-bind variables so that we can pass more complex expressions to functions. Corresponding to the two interpretations of application in \( \mathcal{L}, \mathcal{M} \) provides both lazy let and strict let!. As in \( \mathcal{L} \), the case expression in \( \mathcal{M} \) serves only to force and unpack boxed numbers. In order to be explicit that we must know sizes of variables in \( \mathcal{M} \), we use two different metavariables for \( \mathcal{M} \) variables \( (p \text{ and } i) \), each corresponding to a different kind of machine register.

The \( \mathcal{M} \) language is given an operational semantics in terms of machine states \( \mu \). A machine state is an expression under evaluation, a stack, and a heap. Stacks are an ordered list of stack frames, as explored below; heaps are considered unordered and contain only mappings from pointer variables to expressions. The upper group of rules in Figure 6 apply when the expression is not a value; the rule to use is chosen based on the expression. The lower group of rules apply when the expression is a value; the rule to use is chosen based on the top of the stack.

The first two rules push an argument onto the stack. In the first rule, PAPP, notice that the argument is a variable \( p \) and may not be a value. Evaluating the function first is therefore lazy application. In IAPP, on the other hand, the argument must be a literal and therefore fully evaluated. The stack frames are popped in the first two value rules, which apply when we have fully evaluated the function to expose a \( \lambda \)-expression. In these rules, we use substitution to model function application; in a real machine, of course, parameters to functions would be passed in registers. However, notice that the value being substituted is always of a known width; this substitution is thus implementable.

The VAL rule applies when we are evaluating a variable \( p \) bound to a value in the heap. It does a simple lookup. In contrast, the EVAL rule applies when \( p \) is mapped to a non-value \( t \) (we consider trying VAL before EVAL when interpreting Figure 6). In this case, we proceed by evaluating \( t \). Upon completion (FCE), we then store the value \( t \) reduced to back in the heap; this implements thunk sharing, as performed by GHC.

Lazy let simply adds a mapping to the heap (LET). Strict let!, on the other hand, starts evaluating the let-bound expression \( t_1 \), pushing a continuation onto the stack (SLET). This continuation is then entered when \( t_1 \) has been reduced to a value (ILET). The case expression is similar (CASE), pushing a continuation onto the stack and popping it after evaluation (IMAT).

6.3 Compilation

The languages \( \mathcal{L} \) and \( \mathcal{M} \) are related by the compilation operation, in Figure 7. This type-directed algorithm is parameterized over a variable environment \( V \), containing both mappings from \( \mathcal{L} \)-variables \( x \) to \( \mathcal{M} \)-variables \( y \) as well as a listing of fresh \( \mathcal{M} \)-variables used to compile applications. The \# operator is an assertion that a variable does not appear in \( V \).
Applications are compiled into either lazy or strict `let` expressions, depending on the kind of the argument—this behavior is just as in Haskell and conforms to the two different application rules in L’s operational semantics. Applications of `let` are similarly compiled strictly. Other compilation rules are unremarkable, but we note that compiling an abstraction requires knowing a concrete width for the bound variable.

This compilation algorithm is partial, as it cannot compile, for example, an L-expression that uses levity polymorphism in a variable bound by a λ. The type system of L rules out this possibility. Indeed, L’s type system guarantees that an L-expression can be compiled:

**Theorem (Compilation).** If $\Gamma \vdash e : \tau$ and $\Gamma \Downarrow V$, then $\llbracket e \rrbracket^V_\Gamma \leadsto t$.

The condition $\Gamma \Downarrow V$ requires that $V$ has suitable mappings for the variables bound in $\Gamma$; the full definition appears in the appendix.

The compilation algorithm also critically preserves operational semantics, as proved in this theorem:

**Theorem (Simulation).** Suppose $\Gamma$ has no term variable bindings. If $\Gamma \vdash e : \tau$ and $\Gamma \vdash e \rightarrow e'$, then $\llbracket e \rrbracket^V_\Gamma \leadsto t$, $\llbracket e' \rrbracket^V_\Gamma \leadsto t'$, and $t \rightarrow t'$.

This theorem statement requires the notion of joinability of M-expressions. While the full definition appears in the appendix, intuitively, two M-expressions $t_1$ and $t_2$ are joinable (that is, $t_1 \leftarrow t_2$) when they have a common reduct for any stack and heap. We cannot quite say that $t$ steps to $t'$ in the Simulation Theorem because of the possibility of applications that compile to `let`-bindings, which must be evaluated before we can witness the commonality between $t$ and $t'$.

### 6.4 A missing step

The proof of the Simulation Theorem requires the following technical fact, relating a substitution in L to a substitution in M:

**Assumption (Substitution/compilation).** If:

1. $\Gamma, x : \tau, \Gamma' \vdash e_1 : \tau'$
2. $\Gamma \vdash e_2 : \tau$
3. $\Gamma \vdash \tau : \text{TYPE}$
4. $\llbracket e_1 \rrbracket^V_{\Gamma, x : \tau, \Gamma'} \leadsto t_1$
5. $\llbracket e_2 \rrbracket^V_{\Gamma} \leadsto t_2$

Then there exists $t_3$ such that $\llbracket e_1 \llbracket_{\Gamma, x : \tau, \Gamma'} \rightarrow^* \llbracket e_2 \rrbracket_{\Gamma} \leadsto t_3$ and let $p_2 = t_2 \mid t_1 \llbracket p_2/p \rrbracket \leadsto t_3$, where $p_2$ is fresh.

This assumption is needed when considering the S_BETAPTR rule from L’s operational semantics—we must prove that the reduct and the reduct, with its substitution, compile to joinable M-expressions.

We have not proved this fact, though we believe it to be true.

The challenge in proving this is that the proof requires, in the lazy application case, generalizing the notion of joinability to heaps, instead of just M-expressions. When considering this generalization, we see that it is difficult to write a well-founded definition of joinability, if we consider the possibility of cyclic graphs in the heap.

Interestingly, this assumption is a key part of any proof that compilation from System F to ANF is semantics-preserving. In the functional language compilation community, we have accepted such a transformation for some time. Yet to our surprise, we have been unable to find a proof of its correctness in the literature. We thus leave this step of the correctness proof for the ANF transformation as an open problem, quite separable from the challenge of proving levity polymorphism. Note, in particular, that the assumption

14Lacking recursion, our languages do not support such cyclic structures. However, this restriction surely does not exist in the broader context of Haskell, and it would seem too clever by half to use the lack of recursion in our languages as the cornerstone of the proof.

### 6.5 Conclusion

Following the path outlined at the beginning of this section, we have proved that by restricting the use of levity polymorphism, we can compile a variant of System F that supports levity polymorphism into an ANF language whose operational semantics closely mirrors what would take place on a concrete machine. The compilation is semantics-preserving. This proof shows that our restrictions are indeed sufficient to allow compilation.

### 7. New opportunities from levity polymorphism

#### 7.1 Relaxation of restrictions around unlifted types

Previous to our implementation of levity polymorphism, GHC had to brutally restrict the use of unlifted types:

- **No type family could return an unlifted type.** Recall that previous versions of GHC lumped together all unlifted types into the kind #. Thus the following code would be kind-correct:

  ```haskell
  type family F a :: # where
  F Int = Int#
  F Char = Char#
  ```

  However, GHC would be at a loss trying to compile $f :: F a \rightarrow a$, as there would not be a way to know what size register to use for the argument; the types `Char#` and `Int#` may have different calling conventions. Unboxed tuples all also had kind #, making matters potentially even worse.

- **Unlifted types were not allowed to be used as indices.** It was impossible to pass an unlifted type to a type family or to use one as the index to a GADT. In retrospect, it seems that this restriction was unnecessary, but we had not developed enough of a theory around unlifted types to be sure what was safe. It was safer just to prevent these uses.

- **Unlifted types had to be fully saturated.** There are several parameterized unlifted types in GHC: `Array#` :: `Type` → # is representative. We might imagine abstracting over a type variable a :: `Type` → # and wish to use `Array#` to instantiate a. However, with the over-broad definition of #—which included unlifted types of all manner of calling convention—any such abstraction could cause trouble. In particular, note that $(a :: #) \text{Bool}$ (a partially-applied unboxed tuple) can have type `Type` → #, and its calling convention bears no resemblance to that of `Array#`.

Now that we have refined our understanding of unlifted types as described in this paper, we are in a position to lift all of these restrictions. In particular, note that the F-type family is ill-kinded in our new system, as `Int#` has kind `TYPE` `[IntRep]` while `Char#` has kind `TYPE` `[CharRep]`. Similarly, abstractions over partially-applied unlifted type constructors are now safe, as long as our new, more precise kinds are respected.

#### 7.2 Levity-polymorphic functions

Beyond `error`, `myError` and other functions that never return, there are other functions that can also be generalized to be levity polymorphic. Here is the generalized type of Haskell’s ($) function, which does simple function application:

```haskell
($) :: \forall \,(\tau :: \text{Rep}) \,(a :: \text{Type}) \,(\text{b :: TYPE} \,\tau).
\,(a \rightarrow b) \rightarrow a \rightarrow b
f \$ x = f \,x
```
Note that the argument, \( x \), must have a lifted type (of kind \( \text{Type} \)), but that the return value may be levity-polymorphic, according to the rules in Section 5.1. This generalization of (\$) has actually existed in GHC for some time, due to requests from users, implemented by a special case in the type system. With levity polymorphism, however, we can now drop the special-case code and gain more assurance that this generalization is correct.

We can similarly generalize (\(_\)\), the function composition operator:

\[
\text{(\(_\))} :: \forall (r :: \text{Rep}) (a :: \text{Type}) (b :: \text{Type}) (c :: \text{TYPE} r).
\]

\[
(b \to c) \to (a \to b) \to a \to c
\]

\[
(f \cdot g) x = f (g x)
\]

Once again, we can generalize only the return type. Unlike in the example with (\$), we see that the restriction around levity-polymorphic arguments bites here: this is the only reason that we cannot generalize the kind of \( a \). Also unlike (\$), we had not noticed that it was safe to generalize (\(_\)) in this way. Only by exploring levity polymorphism did this generalization come to light.

### 7.3 Levity-polymorphic classes

Haskell uses type classes to implement ad-hoc polymorphism [15]. An example is the \( \text{Num} \) class, excerpted here:

```haskell
class Num a where
  (+) :: a -> a -> a
  abs :: a -> a
```

Haskell uses the \( \text{Num} \) class to be able to apply numerical operations over a range of numerical types. We can write both \( 3 + 4 \) and \( 2.718 + 3.14 \) with the same (\(+\)) operator, knowing that type class resolution will supply appropriate implementations for (\(+\)) depending on the types of its operands. However, because we have never been able to abstract over unlifted types, unlifted numerical types have been excluded from the convenience of ad-hoc overloading. The library that ships with GHC exports (\(+\)) :: \( \text{Int}_# \to \text{Int}_# \to \text{Int}_# \) and (\(+\)) :: \( \text{Double}_# \to \text{Double}_# \to \text{Double}_# \) in order to perform addition on these two unlifted numerical types. Programmers who wish to use unlifted numbers in their code must use these operators directly.

With levity polymorphism, however, we can extend the type class mechanism to include unlifted types. We generalize the \( \text{Num} \) class thus:

```haskell
class Num (a :: \text{TYPE} r) where
  (+) :: a -> a -> a
  abs :: a -> a
```

The only change is that \( a \) is no longer of type \( \text{Type} \), but can have any associated \( \text{Rep} \). This allows the following instance, for example:

```haskell
instance Num Int_# where
  (+) = (+_#)
  abs n  | n < 0 0# = negateInt_# n
          | otherwise = n
```

We can now happily write \( 3 + 4 \) to add machine integers.\(^{15}\) But how can this possibly work? Let’s examine the type of our new (\(+\)):

\[
(+) :: \forall (r :: \text{Rep}) (a :: \text{TYPE} r).
\]

\( \text{Num} a \Rightarrow a \to a \to a \)

It looks as if (\(+\)) takes a levity-polymorphic argument, something that has been disallowed according to the rules in Section 5.1. Yet we see that nothing untoward happens when we expand out the definitions. Type classes are implemented via the use of dictionaries [6], simple records of method implementations. At runtime, any function with a \( \text{Num} a \) constraint takes a dictionary containing the two methods that are part of our \( \text{Num} \) class. To be concrete, this dictionary type looks like this:

```haskell
data Num (a :: \text{TYPE} r)
  = MkNum \{ (+) :: a -> a -> a, abs :: a -> a \}
```

The methods that the user writes are simply record selectors. The type of the (\(+\)) record selector is the type we see above (blurring the distinction between \( 
\Rightarrow \) and \( 
\to \)), but it’s not properly levity-polymorphic: it takes a \( \text{Num} a \) dictionary (a lifted type) and returns a function of type \( a \to a \to a \). All functions are lifted, so there is no improper levity polymorphism at play.

When the user writes an instance, GHC translates each method implementation to a top-level function. Let’s call the functions \( \text{plusInt}_# \) and \( \text{absInt}_# \). They are fully monomorphic, taking and returning \( \text{Int}_# \) values; there is no levity polymorphism there, so they cannot run afoot of our restrictions.

With these functions defined, GHC then builds the dictionary, thus:

\[
\text{\$d} :: \text{Num Int}_#
\]

\[
\text{\$d} = \text{MkNum} \{ (+) = \text{plusInt}_#, \text{abs} = \text{absInt}_# \}
\]

Once again, there is nothing unpleasant here—this snippet is indeed entirely monomorphic.

When we put this all together, we can see that this use of levity polymorphism is all acceptable. We are treading close to the cliff, however. Consider this:

\[
\text{abs1}, \text{abs2} :: \forall (r :: \text{Rep}) (a :: \text{TYPE} r).
\]

\[
\text{Num} a \Rightarrow a \to a
\]

\[
\text{abs1} = \text{abs}
\]

\[
\text{abs2} x = \text{abs} x
\]

The definition for \( \text{abs1} \) is acceptable; there are no levity-polymorphic bound variables. However, \( \text{abs2} \) is rejected! It binds a levity-polymorphic variable \( x \). And yet \( \text{abs2} \) is clearly just an \( \eta \)-expansion of \( \text{abs1} \). How can this be possible?

When we consider how a function is compiled, it becomes clearer. Despite the currying that happens in Haskell, a compiled function is assigned an \( \eta \)-arity, declaring how many parameters it accepts via the machine’s usual convention for passing parameters to a function. The \( \text{abs1} \) function has an \( \eta \)-arity of 1: its one parameter is the \( \text{Num} a \) dictionary (which, recall, is a perfectly ordinary value of a lifted type). It returns the memory address of a function that takes one argument. On the other hand, \( \text{abs2} \) has an \( \eta \)-arity of 2, taking also the levity-polymorphic value to operate on and returning a levity-polymorphic value. It is this higher \( \eta \) arity that causes trouble for \( \text{abs2} \). When compiling, \( \eta \)-equivalent definitions are not equivalent!

### 8. Polymorphism in other languages

#### 8.1 Cyclone

One can view the kind system we describe in this paper as a generalization of the idea originally put forth as a part of Cyclone [7]. Cyclone was conceived as a type-safe C-like language. The designers of Cyclone naturally wanted the language to support parametric polymorphism; indeed, this feature is the core subject of a journal-length article on the language [5]. However, parametric polymorphism in a language that supports values of differing sizes will necessarily crash headlong into the same problem we describe here: it is impossible to compile a function whose parameters have a width unknown at compile time. Grossman [5 Section 4.1] describes a
kind system for Cyclone supporting two kinds, $A$ (for “any”) and $B$ (for “boxed”). This kind system supports sub-kinding, where $B <: A$. In this system, all pointers have kind $B$, as well as int; all types have kind $A$. Accordingly, the use of abstract types (such as type variables) of kind $A$ is restricted to contexts that do not need to know the width of the type. Such types can appear in pointer types; for example, if type variable $\alpha$ has kind $A$, then a variable of type $\alpha^*$ (that is, a pointer to $\alpha$) is allowed, but a variable of type $\alpha$ itself is not. Cyclone’s restrictions around types of kind $A$ have the same flavor of our restrictions around levity-polymorphic types.

Indeed, we can view Cyclone’s $A$ roughly as our $\forall \ (r :: Rep). TYPE r$. However, by being more explicit about representation quantification, we can use the same $r$ variable in multiple kinds, leading to the applications in Section 7. Our kind system also allows for more than one concrete kind, unlike Cyclone’s.

8.2 C++

C++’s approach to parametric polymorphism is its template system. A C++’s template is essentially a macro, where the type at which the template is to be specialized is inferred at use sites. Consider the templated identity function:

```
template<typename T> T id(T t) { return t; }
```

If we were now to say `id(5)`, the C++ compiler would specialize that function, producing `int id(int t) { return t; }`. Calling `id(“x”)` would produce a similar specialization to characters. In effect, polymorphism is eliminated at compile time, via specialization. Hence C++ need not worry about the compilation problems we describe in this paper; only concrete instantiations are compiled.

One challenge with this approach is that templates must get specialized independently in each compilation unit. The linker then deduplicates when combining the object files. Because of this, compile times with templates can be slower, or compilers have to design elaborate caching mechanisms to avoid recompilation. Another major drawback is that this approach cannot support higher-rank types or polymorphic recursion, both widely used by Haskell programmers.

8.3 Java

Java’s generics [11] provide parametric polymorphism. This feature takes the approach Haskell has taken before our contributions, in requiring every type argument to a polymorphic function to be boxed. In the Java terminology, reference types are boxed, while primitive types are unboxed. In order to better support polymorphic functions and classes, Java provides a boxed type corresponding to every unboxed type and inserts conversions as necessary. While these conversions are invisible to the programmer, they must be executed at runtime, with the attendant time and space costs.

Java has no support for user-defined unboxed types, so it is conceivable for a programmer to manually specialize any polymorphic definition to all available primitive types—there are only eight. Indeed, the standard library does exactly this for many definitions.

8.4 OCaml

OCaml’s treatment of parametric polymorphism is unsurprisingly very similar to Haskell’s, but with one major difference: OCaml’s `int` type is unboxed, as well as any type that can be packed into an `int`, like `char` or enumerations [10]. However, because the width of an `int` is always the same as the width of a pointer, polymorphic functions in OCaml can be used at type `int*`. Beyond those types that can be packed into an `int`, OCaml does not have first-class unboxed types.

8.5 C# / .NET

The .NET generics system allows unrestricted parametric polymorphism over unboxed types, by taking advantage of the JIT to generate specialized code at runtime, for each distinct runtime representation type [8, Section 4]. These specialized versions are generated lazily, on demand, and the code is shared if the types are “compatible”. As the paper says: “Two instantiations are compatible if for any parameterized class its compilation at these instantiations gives rise to identical code and other execution structures.”

To our knowledge, .NET is the only widely-used language implementation that supports unrestricted polymorphism over unboxed types, a truly impressive achievement. Runtime types must be passed to every polymorphic function, but this is needed for other reasons too (e.g., reflection).

8.6 Rust

Although undocumented in the research literature, Rust’s generics appear quite like C++’s templates, with monomorphism upon request. Rust’s compiler avoids compilation overhead by compiling a generic definition into an intermediate form, ready for specialization. This approach means that polymorphic definitions can be used at both unboxed and boxed types but encounters the same restrictions around polymorphic recursion and higher-rank types that C++ does.

9. Conclusion

This paper presents a new way to understand the limits of parametric polymorphism, claiming that kinds are calling conventions. We thus must fix the kind of any bound variables and arguments before we can compile a function. Even with these restrictions, however, we find that our novel levity polymorphism—the ability to abstract over a type’s runtime representation—is practically useful and extends the expressiveness of Haskell. Furthermore, we have proved our restrictions to be sufficient to allow compilation and have implemented our ideas in a production compiler. It is our hope that this new treatment of polymorphism can find its way to new languages, several of which currently exhibit a number of compromises around polymorphism.

References


A. Proofs

A.1 Preliminaries

Definition 1 (Garbage collection). Define gc(p) to do garbage collection on a machine state. This removes any Force(p) stack elements and any p → t heap bindings for pointers p which are not reachable from the expression in p nor in App(p), Let(y, t), or Case(y, t) stack elements.

Definition 2 (Equivalent states). Two machine states µ1 and µ2 are equivalent, written µ1 ≡ µ2 if gc(µ1) = gc(µ2).

Definition 3 (Joinability). Two M-expressions t1 and t2 are joinable, written t1 ≻ t2, when there exists t3 such that, for all S and H, there exist S1, H1, S2, and H2 such that:

1. (t1; S; H) →∗ (t1; S; H1);
2. (t2; S; H) →∗ (t2; S; H2);
3. (t3; S1; H1) (t3; S2; H2); and
4. gc((t3; S; H1)) = (t3; S'; S; H'), where H ⊆ H'.

In the above definition, note that the last line states that the garbage-collected stack of the common reduct machine state must match the original stack.

Assumption 4 (Uniqueness of variables). We assume that all contexts Γ bind any variable at most once.

A.2 Determinacy

The following determinacy lemmas are all proven by straightforward induction.

Lemma 5 (Kind determinacy). If Γ ⊢ τ : κ1 and Γ ⊢ τ : κ2, then κ1 = κ2.

Lemma 6 (Type determinacy). If Γ ⊢ e : τ1 and Γ ⊢ e : τ2, then τ1 = τ2.

Lemma 7 (Reduction determinacy). If Γ ⊢ e → e1 and Γ ⊢ e → e2, then e1 = e2.

Lemma 8 (Compilation determinacy). If [e]V ⊢ t1 and [e]V' ⊢ t2, then t1 = t2.


A.3 Weakening

Lemma 9 (Typing weakening). If Γ ⊢ e : τ and Γ' = Γ1, Γ2, then Γ' ⊢ e : τ.

Proof. Straightforward induction on the derivation of Γ ⊢ e : τ.

Lemma 10 (Variable assignment weakening). If [e]V ⊢ t, Γ' = Γ1, Γ2, and V' = V1, V2, then [e]V' ⊢ t.


A.4 Substitution

The following substitution lemmas are all proved by the standard approach for substitution lemmas, with multiple cases for variables and induction elsewhere. Later lemmas depend on earlier ones.

Lemma 11 (Representation substitution in kinds). If Γ, r, Γ' ⊢ τ kind and fv(ρ) ⊆ Γ, then Γ, Γ'ρ[ρ/r] ⊢ τρ[ρ/r] kind.

Lemma 12 (Representation substitution in types). If Γ, r, Γ' ⊢ τ : κ and fv(ρ) ⊆ Γ, then Γ, Γ'ρ[ρ/r] ⊢ τρ[ρ/r] : κρ[ρ/r]

Lemma 13 (Representation substitution in expressions). If Γ, r, Γ' ⊢ e : τ and fv(ρ) ⊆ Γ, then Γ, Γ'ρ[ρ/r] ⊢ eρ[ρ/r] : τρ[ρ/r].

Lemma 14 (Type substitution in types). If Γ, α:κ', Γ' ⊢ τ : κ and Γ ⊢ τ' : κ', then Γ, Γ'ρ[ρ/α] ⊢ τρ[ρ/α] : κρ[ρ/α].

Lemma 15 (Type substitution in terms). If Γ, τ':κ, Γ' ⊢ e : τ and Γ ⊢ τ' : κ, then Γ, Γ'ρ[ρ/α] ⊢ eρ[ρ/α] : τρ[ρ/α].

Lemma 16 (Substitution). If Γ, x:τ', Γ' ⊢ e : τ and Γ ⊢ e' : τ', then Γ, Γ' ⊢ e[e'/x] : τ.

A.5 Preservation

Theorem 17 (Preservation). If Γ ⊢ e : τ and Γ ⊢ e → e', then Γ ⊢ e' : τ.

Proof. By induction on the derivation of Γ ⊢ e : τ.

Case E_VAR: Impossible, as variables do not step.

Case E_APP: We now have several cases, depending on how e has stepped:

Case S_APPLET: By Lemma 16
Case S_APPBETA: By Lemma 16
Case S_APPSTRONG: By induction hypothesis.
Case S_APPSTRONG2: By the induction hypothesis.
Case S_BETAUNBOXED: By Lemma 16
Case E.LAM: Impossible.
Case E.TLAM: We must step by S.TLAM. We are done by the induction hypothesis.
Case E.TAPP: We now have two cases, depending on how $e$ has stepped:
  - Case S.TAPP: By the induction hypothesis.
  - Case E.RBETA: By Lemma $[15]$
Case E.RLAM: We must step by S.RLAM. We are done by the induction hypothesis.
Case E.RAPP: Like the previous case, but appealing to Lemma $[13]$
Case E.CON: We must step by S.CON. We are done by the induction hypothesis.
Case E.CASE: We now have two cases, depending on how $e$ has stepped:
  - Case S.CASE: By the induction hypothesis.
  - Case S.MATCH: By Lemma $[16]$
Case E.ERROR: Trivial.
Literal cases: Impossible.

A.6 Progress

Lemma 18 (Canonical forms). Suppose $\Gamma$ has no term variable bindings and $\Gamma \vdash v : \tau$.

1. If $\tau = \tau_1 \rightarrow \tau_2$, then $v = \lambda x : \tau_1. \ e$.
2. If $\tau = \forall \alpha : \kappa. \tau_0$, then $\tau = \lambda \alpha : \kappa. \ tau_0$.
3. If $\tau = \forall \tau_0$, then $v = \Lambda \tau. \ \tau_0$.
4. If $\tau = \text{Int}$, then $v = I_0[n]$.
5. If $\tau = \text{Int}_\text{Nat}$, then $v = n$.
6. $\tau$ is not a type variable.

Proof. By induction on the derivation of $\Gamma \vdash v : \tau$.

Theorem 19 (Progress). Suppose $\Gamma$ has no term variable bindings. If $\Gamma \vdash e : \tau$, then either $\Gamma \vdash e \rightarrow e'$ or $e$ is a value.

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$.

Case E.VAR: Impossible with no bound term variables.
Case E.APP:

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_2 : \tau_1 \\
\hline
\Gamma \vdash \tau_1 : \text{TYPE} v \\
\hline
\Gamma \vdash e_1 \ e_2 : \tau_2 \\
\end{array}
\]

E_APP

We have three cases, depending on the value of $v$.

Case $v = \text{P}$: Use the induction hypothesis on $e_1$, yielding two cases:
  - Case $\Gamma \vdash e_1 \rightarrow e'_1$: We are done by S_APPLAZY.
  - Case $e_1$ is a value: Lemma $[15]$ tells us that $e_1 = \lambda x : \tau_1. \ e_0$. We are done by S_BETAAPTR.

Case $v = \text{I}$: Use the induction hypothesis on $e_1$, yielding two cases:
  - Case $\Gamma \vdash e_2 \rightarrow e'_2$: We are done by S_APPEQUAL.
  - Case $e_2$ is a value: Now we use the induction hypothesis on $e_1$, yielding two cases:
    - Case $\Gamma \vdash e_1 \rightarrow e'_1$: We are done by S_BETAAPTR.
    - Case $e_1$ is a value: Lemma $[19]$ tells us that $e_1 = \lambda x : \tau_1. \ e_0$ and thus we are done by S_BETAAPTR.

Case E.LAM: Here, $e$ is a value.
Case E.TLAM: We have learned that $e = \lambda \alpha : \kappa. \ e_0$. Using the induction hypothesis on $e_0$ gives us two possibilities:
  - Case $\Gamma \vdash e_0 \rightarrow e'_0$: We are done by S_TLAM.
  - Case $e_0$ is a value: Then $e$ is a value.
Case E.TAPP: We have learned that $e = e_0 \tau$. Using the induction hypothesis on $e_0$ gives us two possibilities:
  - Case $\Gamma \vdash e_0 \rightarrow e'_0$: We are done by S_TAPP.
  - Case $e_0$ is a value: Lemma $[18]$ tells us that $e_0 = \lambda \alpha : \kappa. \ e_0$ and then we are done by S_TBETA.
Case E.RLAM: Like the case for E.TLAM, using S_RLAM.
Case E.RAPP: Like the case for E.RAPP, using S_RAPP and S_RBETA.
Case E.CON: We learn that $e = I_0[e_0]$. Using the induction hypothesis on $e_0$ gives us two possibilities:
  - Case $\Gamma \vdash e_0 \rightarrow e'_0$: We are done by S_CON.
  - Case $e_0$ is a value: Then $e$ is a value.
Case E.CASE: We learn that $e = \text{case } e_1 \text{ of } I_0[x] \rightarrow e_2$. Using the induction hypothesis on $e_1$ gives us two possibilities:
  - Case $\Gamma \vdash e_1 \rightarrow e'_1$: We are done by S_CASE.
  - Case $e_1$ is a value: We see that $\Gamma \vdash e_1 : \text{Int}$, and then Lemma $[18]$ tells us that $e_1 = I_0[n]$. We are done by S_MATCH.
Case E.ERROR: We are done by S_ERROR.
Case E_INTLit: Here, \( e \) is a value.

A.7 Compilation

\[ \Gamma \propto V \]

**Compatibility**

\[ \compl{\emptyset} \]

\[ \Gamma \propto V \quad \compl{\text{Fresh}} \]

\[ \Gamma \propto V, y \quad \compl{\text{TYVar}} \]

\[ \Gamma, \alpha : \kappa \propto V \quad \compl{\text{RepVar}} \]

\[ \Gamma, \tau \propto V \quad \compl{P} \]

\[ \Gamma \propto V, x \mapsto p \quad \compl{I} \]

**Lemma 20 (Variable assignments).** If \( \Gamma \propto V \) and \( x : \tau \in \Gamma \), then \( x \mapsto y \in V \).

**Proof.** Straightforward induction on the derivation of \( \Gamma \propto V \).

**Theorem 21 (Compilation).** If \( \Gamma \vdash e : \tau \) and \( \Gamma \propto V \), then \( e \Vdash t \).

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

Case E_VAR: By Lemma 20 and C_VAR.

Case E_APP:

\[ \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \]
\[ \Gamma \vdash e_2 : \tau_1 \]
\[ \Gamma \vdash \tau_1 : \text{TYPE} \neg \]
\[ \Gamma \vdash e_1 : \tau_2 \]
\[ \Gamma \vdash \text{E_APP} \]

Inversion tells us that \( \Gamma \vdash e_2 : \tau_1 \) and that \( \Gamma \vdash \tau_1 : \text{TYPE} \neg \). We now have two cases, depending on the value of \( \neg \): 

Case \( \neg = P \): Proceed by C_APPLazy. Choose \( p \) such that \( p \# V \). We see that \( \left[ e_1 \right]_p \rightarrow \tau_1 \) and \( \left[ e_2 \right]_p \rightarrow \tau_2 \) by induction and Lemma 10.

Case \( \neg = \text{Int} \): Similar, by C_APPINT.

Case E_LAM:

\[ \Gamma, x : \tau_1 \vdash e : \tau_2 \]
\[ \Gamma \vdash \tau_1 : \text{TYPE} \neg \]
\[ \Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \]
\[ \Gamma \vdash \text{E_LAM} \]

We have two cases, depending on the value of \( \neg \):

Case \( \neg = P \): By C_LAM_PTR and the induction hypothesis.

Case \( \neg = \text{Int} \): By C_LAM_INT and the induction hypothesis.

Case E_TLAM: By C_TLAM and the induction hypothesis.

Case E_TAPP: By C_TAPP and the induction hypothesis.

Case E_RLAM: By C_RLAM and the induction hypothesis.

Case E_RAPP: By C_RAPP and the induction hypothesis.

Case E_CON:

\[ \Gamma \vdash e : \text{Int} \neg \]
\[ \Gamma \vdash t_p[e] : \text{Int} \quad \text{E_CON} \]

Inversion tells us that \( \Gamma \vdash e : \text{Int} \neg \). We can thus proceed by C_CON, Lemma 10 and the induction hypothesis.

Case E_CASE:

\[ \Gamma \vdash e_1 : \text{Int} \]
\[ \Gamma \vdash e_2 : \tau \]
\[ \Gamma, x : \text{Int} \neg \vdash \text{case } e_1 \text{ of } t_p[x] \rightarrow e_2 : \tau \quad \text{E_CASE} \]

By the induction hypothesis and C_CASE.

Case E_ERROR: By C_ERROR.

Case E_INTLit: By C_INTLit.
A.8 Simulation

Assumption 22 (Substitution/compilation). If:

1. \( \Gamma, \alpha, \tau, \Gamma' \vdash e_1 : \tau' \)
2. \( \Gamma \vdash e_2 : \tau \)
3. \( \Gamma \vdash \tau : \text{TYPE} \)
4. \( \left[ e_2 \right]_{\Gamma, x : \tau, \Gamma'} \dashv t_1 \) and
5. \( \left[ e_2 \right]_\Gamma \dashv t_2 \)

Then there exists \( t_3 \) such that \( \left[ e_1[e_2/x] \right]_{\Gamma, x : \tau, \Gamma'} \dashv t_3 \) and let \( p_2 = t_2 \in t_1[p_2/p] \Leftrightarrow t_3 \), where \( p_2 \) is fresh.

See Section 6.4 for why this is an assumption.

Lemma 23 (Int_{\#} substitution/compilation). If \( \left[ e \right]_{\Gamma, x : \tau, \Gamma'} \vdash t \) and \( \left[ v \right]_\Gamma \vdash n \), then \( \left[ e[v/x] \right]_{\Gamma, x : \tau, \Gamma'} \vdash t[n/i] \).

Proof. By induction on the derivation of \( \left[ e \right]_{\Gamma, x : \tau, \Gamma'} \vdash t \).

Case C_VAR: We have two cases:

Case \( e = x \): In this case, \( t = i \), \( e[v/x] = v \), and \( t[n/i] = n \). We are done by the assumption that \( \left[ v \right]_\Gamma \vdash n \) and Lemma 10.

Case \( e \neq x \): The substitutions are no-ops, and so we are done.

Other rules: By induction (and Lemma 10) in cases with typing rule premises.

\[ \square \]

Lemma 24 (Type substitution/compilation). If \( \left[ e \right]_{\Gamma, \alpha, \tau, \Gamma'} \vdash t \), then \( \left[ e[\tau/\alpha] \right]_{\Gamma, \alpha, \tau, \Gamma'} \vdash t \).

Proof. By induction on the derivation of \( \left[ e \right]_{\Gamma, \alpha, \tau, \Gamma'} \vdash t \), appealing to Lemma 14 and Lemma 15 in the cases with typing premises.

\[ \square \]

Lemma 25 (Representation substitution/compilation). If \( \left[ e \right]_{\Gamma, \tau, \Gamma'} \vdash t \), then \( \left[ e[\rho/\tau] \right]_{\Gamma, \tau, \Gamma'} \vdash t \).

Proof. By induction on the derivation of \( \left[ e \right]_{\Gamma, \tau, \Gamma'} \vdash t \), appealing to Lemma 12 and Lemma 13 in the cases with typing premises.

\[ \square \]

Lemma 26 (Canonical runtime forms). Let \( \Gamma \) be a typing context with no bound term variables, and suppose \( \Gamma \vdash v : \tau \) and \( \Gamma \vdash \tau : \text{TYPE} \). Then \( \left[ v \right]_\Gamma \vdash n \) for some \( n \).

Proof. Proceed by induction on the derivation of \( \Gamma \vdash \tau : \text{TYPE} \).

Case T_INT: Impossible.
Case T_INTH: Lemma 18 tells us that \( v = n \). We are done by C_INTLIT.
Case T_ARROW: Impossible.
Case T_VAR: Lemma 18 tells us this is impossible.
Case T_ALLTY: We learn that \( \tau = \forall \alpha : \tau_0 \). Lemma 18 tells us that \( v = \lambda \alpha : n. v_0 \). Inversion then tells us that \( \Gamma, \alpha : \tau_0 \vdash v_0 : \tau_0 \) and that \( \Gamma, \alpha : \tau_0 \vdash \tau_0 : \text{TYPE} \). The induction hypothesis tells us that \( \left[ v_0 \right]_{\Gamma, \alpha : \tau_0} \vdash n \). We are done by C_TLAM.
Case T_ALLREP: Similar to previous case, using C_RLAM.

\[ \square \]

Theorem 27 (Simulation). Suppose \( \Gamma \) has no term variable bindings. If \( \Gamma \vdash e : \tau \) and \( e \to e' \), then \( \left[ e \right]_\Gamma \vdash t \), \( \left[ e' \right]_\Gamma \vdash t' \), and \( t \equiv t' \).

Proof. By induction on the derivation of \( \Gamma \vdash e : \tau \). Unless stated otherwise, assume that \( t \) variables are the compilations of the corresponding \( e \) variables. Note that the existence of such \( t \) is guaranteed by inversion and Theorem 21.

Case E_VAR: Impossible.
Case E_APP: Here, \( e = e_1 e_2 \). We now have several cases, depending on how \( e \) has stepped:

Case S_APPLAZY:

\[
\begin{align*}
\Gamma \vdash e_2 : \tau \\
\Gamma \vdash e_1 \to e'_1 \\
\Gamma \vdash \tau : \text{TYPE} \nonumber \\
\text{S_APPLAZY} \\
\Gamma \vdash e_1 e_2 \to e'_1 e_2 \nonumber
\end{align*}
\]

The induction hypothesis tells us that \( \left[ e'_1 \right]_\Gamma \vdash t'_2 \) and that \( t_1 \) and \( t'_2 \) have a common reduct \( t'_1 \). The output of the C_APPLAZY rule is let \( p = t_2 \in t_1 p \). We can see that

\[
\begin{align*}
\{\text{let } p = t_2 \in t_1 p; S; H\} & \to \langle t_1 p; S; p \to t_2, H \rangle \\
& \to \langle t_1; \text{App}(p), S; p \to t_2, H \rangle \\
& \to^\ast \langle t'_2; \text{App}(p), S'; S; p \to t_2, H' \rangle
\end{align*}
\]
We also see that $[e_1, e_2]_\Gamma^0 \leadsto \text{let } p = t_2 \text{ in } t_1' \text{ p and that}$
\[
\begin{align*}
\{\text{let } p = t_2 \text{ in } t_1'; p; S; H\} & \rightarrow (t_1'; p; S; p \mapsto t_2, H) \\
& \rightarrow (t_1'; \text{App}(p), S; p \mapsto t_2, H) \\
& \rightarrow (t_1'; \text{App}(p), S', S; p \mapsto t_2, H')
\end{align*}
\]

We have thus shown that the compilation of $e_1, e_2$ and that of $e_1', e_2'$ have a common reduct, and so we are done with this case.

Case S_BetaPtr:

\[
\Gamma \vdash \tau : \text{TYPE P} \\
\Gamma, (\lambda x : \tau. e_1) e_2 \rightarrow e_1 [e_2/x] \quad \text{S_BetaPtr}
\]

Here, we learn that $e_1$ is $\lambda x : \tau_1. e_1'$. Thus, $t_1$, the compilation of $e_1$, must be by C_LamPtr and must equal $\lambda p_2, t_1'$ where we know $\llbracket e_1 \rrbracket_{\Gamma, x : \tau_1} \leadsto t_1'$ by Theorem 21. We can see that
\[
\begin{align*}
\{\text{let } p = t_2 \text{ in } (\lambda p_2, t_1') p; S; H\} & \rightarrow ((\lambda p_2, t_1') p; S; p \mapsto t_2, H) \\
& \rightarrow (\lambda p_2, t_1' \text{ App}(p), S; p \mapsto t_2, H) \\
& \rightarrow (t_1'[p/p_2]; S; p \mapsto t_2, H)
\end{align*}
\]
The $\beta$-reduct is $e_1'[e_2/x]$. Assumption 22 gives us $t_3$ such that $[e_1'[e_2/x]]_\Gamma^0 \leadsto t_3$ and let $p_3 = t_2 \text{ in } t_3'[p_3/p_2] \leadsto t_3$. We can see that
\[
\begin{align*}
\{\text{let } p_3 = t_2 \text{ in } t_3'[p_3/p_2]; S; H\} & \rightarrow (t_3'[p_3/p_2]; S; p_3 \mapsto t_2, H) \\
& \rightarrow (t_3'[p_3/p_2]; S; p_3 \mapsto t_2, H)
\end{align*}
\]

This final machine state is $\alpha$-equivalent to the final machine state reached earlier. We are thus done with this case, as this state is joinable with $t_3$, the compilation of the $\beta$-reduct.

Case S_AppStrict:

\[
\begin{align*}
\Gamma \vdash e_2 : \tau & \quad \Gamma \vdash \tau : \text{TYPE I} \\
\Gamma \vdash e_2 \rightarrow e_2' & \quad \Gamma \vdash e_1 e_2 \rightarrow e_1 e_2' \quad \text{S_AppStrict}
\end{align*}
\]

The induction hypothesis tells us that $\llbracket e_2 \rrbracket_\Gamma^0 \leadsto t_2'$ and that $t_2$ and $t_2'$ have a common reduct $t_3'$. The output of C_AppInt is $\text{let}! \ i = t_2 \text{ in } t_1 \ i$. We can see that
\[
\begin{align*}
\{\text{let}! \ i = t_2 \text{ in } t_1 \ i; S; H\} & \rightarrow (t_2; \text{Let}(i, t_1 \ i), S; H)
\end{align*}
\]

Compiling the reduct $e_1 e_2'$ yields (using Theorem 17 to show that $e_2'$ has the same type as $e_2$) $\text{let}! \ i = t_2' \text{ in } t_1 \ i$. We can see that
\[
\begin{align*}
\{\text{let}! \ i = t_2' \text{ in } t_1 \ i; S; H\} & \rightarrow (t_2'; \text{Let}(i, t_1 \ i), S; H)
\end{align*}
\]

Both of the resulting machine states must eventually reduce to (after garbage collection) $(t_2'; S', \text{Let}(i, t_1 \ i), S; H')$ (for some $H'$ with $H \subseteq H'$) and so we are done with this case.

Case S_AppStrict2:

\[
\begin{align*}
\Gamma \vdash v_2 : \tau & \quad \Gamma \vdash \tau : \text{TYPE I} \\
\Gamma \vdash e_1 \rightarrow e_1' & \quad \Gamma \vdash e_1 v_2 \rightarrow e_1' v_2 \quad \text{S_AppStrict2}
\end{align*}
\]

Lemma 26 tells us that $\llbracket v_2 \rrbracket_\Gamma^0 \leadsto n_2$. Compiling the reduct (by C_AppInt) yields $\text{let}! \ i = n_2 \text{ in } t_1 \ i$. We can see that
\[
\begin{align*}
\{\text{let}! \ i = n_2 \text{ in } t_1 \ i; S; H\} & \rightarrow (n_2; \text{Let}(i, t_1 \ i), S; H) \\
& \rightarrow ((t_1 \ i)[n_2/i]; S; H) \\
& = (t_1 \ n_2; S; H) \\
& \rightarrow (t_1; \text{App}(n_2), S; H)
\end{align*}
\]

Compiling the reduct (also by C_AppInt) yields $\text{let}! \ i = n_2 \text{ in } t_1' \ i$. Starting in a machine state with $S$ and $H$, this similarly reduces to $(t_1'; \text{App}(n_2), S; H')$. The induction hypothesis tells us that $t_1$ and $t_1'$ are joinable, and so we are done.

Case S_BetaUnboxed:

\[
\Gamma \vdash \tau : \text{TYPE I} \\
\Gamma \vdash (\lambda x : \tau. e) v \rightarrow e[v/x] \quad \text{S_BetaUnboxed}
\]

Here, we learn that $e_1$ is $\lambda x : \tau. e_1'$. Thus, $e_1$, the compilation of $e_1$, must be by C_LamInt and must equal $\lambda i. t_1'$ where we know $\llbracket e_1 \rrbracket_{\Gamma, x : \tau} \leadsto t_1'$ by Theorem 21. We can see that
\[
\begin{align*}
\{\text{let}! \ i = n \text{ in } (\lambda i_2, t_1') i; S; H\} & \rightarrow (n; \text{Let}(i, (\lambda i_2, t_1') i), S; H) \\
& \rightarrow (((\lambda i_2, t_1') i)[n/i]; S; H) \\
& = ((\lambda i_2, t_1') n; S; H) \\
& \rightarrow (\lambda i_2, t_1' \text{ App}(n), S; H) \\
& \rightarrow (t_1'[n/i_2]; S; H)
\end{align*}
\]
The $\beta$-reduct is $e'_1[v/x]$. Lemma \[22\] tells us that $[e'_1[v/x]]^M \sim t'_1[n/i_2]$, which is the same as the final term in the derivation above. We are done with this case.

Case E_LAM: Impossible, as $\lambda$-terms do not step.

Case E_TLAM: We know here that $e = \Lambda\alpha:\kappa. e_0$ and that it has stepped by S_TLAM to $\Lambda\alpha:\kappa. e'_0$. According to C_TLAM, we are done by the induction hypothesis.

Case E_TAPP: We now have two cases, depending on how $e$ has stepped:

Case S_TAPP: We learn here that $e = e_1 \tau$ and that $\Gamma \vdash e_1 \rightarrow e'_1$. Both $e_1 \tau$ and $e'_1 \tau$ compile via C_TAPP, yielding the same $\mathcal{M}$-expression as compiling $e_1$ or $e'_1$ respectively. We are done by induction.

Case S_TBETA: We learn here that $e = (\Lambda\alpha:\kappa. e_1) \tau$. This compiles via C_TAPP and C_TLAM. Inversion and Theorem \[21\] tells us $[e_1]^{\mathcal{M}} \sim t_1$ and therefore that $[(\Lambda\alpha:\kappa. e_1) \tau]^{\mathcal{M}} \sim t_1$. The $\beta$-reduct is $e_1[\tau/\alpha]$. Lemma \[22\] tells us that $[e_1[\tau/\alpha]]^{\mathcal{M}} \sim t_1$, and we are done.

Case E_RLAM: Similar to the case for E_TLAM, using C_RLAM.

Case E_RAPP: Similar to the case for E_TAPP, appealing to Lemma \[25\] in the S_RBETA case.

Case E_CON: We know here that $e = l_\#[e_0]$ and that $e$ has stepped via S_CON; thus $\Gamma \vdash e_0 \rightarrow e'_0$. The reduct compiles via C_CON to let! $i = t_0$ in $l_\#[i]$. We can see that

$$(\text{let}! \ i = t_0 \ \text{in} \ l_\#[i]; S; H) \rightarrow (t_0; \text{let}(i, l_\#[i]), S; H)$$

The reduct similarly compiles to let! $i = t'_0$ in $l_\#[i]$, and we see that

$$(\text{let}! \ i = t'_0 \ \text{in} \ l_\#[i]; S; H) \rightarrow (t'_0; \text{let}(i, l_\#[i]), S; H)$$

We are thus done by the induction hypothesis, relating $t_0$ and $t'_0$.

Case E_CASE: We learn here that $e = \text{case } e_1 \ \text{of } l_\#[x] \rightarrow e_2$. We have two cases, depending on how $e$ has stepped.

Case S_CASE: Here, we know $\Gamma \vdash e_1 \rightarrow e'_1$. The reduct compiles via C_CASE to case $t_1$ of $l_\#[i]$ to $t_2$. We can see that

$$(\text{case } t_1 \ \text{of } l_\#[i] \rightarrow t_2; S; H) \rightarrow (t_1; \text{Case}(i, t_2), S; H)$$

The reduct compiles also via C_CASE to case $t'_1$ of $l_\#[i]$ to $t_2$. We can see that

$$(\text{case } t'_1 \ \text{of } l_\#[i] \rightarrow t_2; S; H) \rightarrow (t'_1; \text{Case}(i, t_2), S; H)$$

These last two are joinable by the induction hypothesis, so we are done.

Case S_MATCH: In this case, we know that $e_1 = l_\#[n]$. Thus the reduct compiles to case $l_\#[n]$ of $l_\#[i]$ to $t_2$, and we can see that

$$(\text{case } l_\#[n] \ \text{of } l_\#[i] \rightarrow t_2; S; H) \rightarrow (l_\#[n]; \text{Case}(i, t_2), S; H) \rightarrow (t_2[n/i]; S; H)$$

The reduct is $e_2[n/x]$. By Lemma \[23\] we see that $[e_2[n/x]]^{\mathcal{M}} \sim t_2[n/i]$, and we are done.

Case E_ERROR: Our expression $e$ must step by S_ERROR and be compiled by C_ERROR. We note that $\langle \text{error}; S; H \rangle \rightarrow \bot$ and we are done.

Case E_INTLIT: Impossible.

□