2017

Levity Polymorphism (extended version)

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

Simon Peyton Jones
Microsoft Research Cambridge

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 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:

$$\begin{aligned} &bTwice :: \forall\ a.\ Bool \to a \to (a \to a) \to a \\ &bTwice\ b\ x\ f = \textbf{case}\ b\ \textbf{of}\ True \to f\ (f\ x) \\ &\qquad\qquad\qquad\qquad\quad False \to x \end{aligned}$$

The function is *polymorphic*[1] 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. Our 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 [5] 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[2] 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 $1 \ldots n$:

[1] We use the term polymorphism to refer exclusively to *parametric* polymorphism.

[2] GHC extends Haskell in many ways to better support high-performance code, so when we say "Haskell" we will always mean "GHC Haskell"

	Boxed	Unboxed
Lifted	*Int* *Bool*	
Unlifted	$ByteArray_\#$	$Int_\#$ $Char_\#$

Figure 1. Boxity and levity, with examples

$$
\begin{aligned}
&sumTo :: Int \to Int \to Int \\
&sumTo\ acc\ 0 = acc \\
&sumTo\ acc\ n = sumTo\ (acc + n)\ (n - 1)
\end{aligned}
$$

GHC represents a values of type *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 *Int*. If *sumTo* used this representation throughout, it would be unbearably slow. Each iteration would evaluate its second argument,[3] follow the the pointer to get the value, and test it against zero; in the non-zero case it would allocate thunks for $(acc + n)$ and $(n - 1)$, 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 $Int_\#$ of unboxed integers [12]. An $Int_\#$ is represented not by a pointer but by the integer itself. Now we can rewrite *sumTo* like this[4]

$$
\begin{aligned}
&sumTo_\# :: Int_\# \to Int_\# \to Int_\# \\
&sumTo_\#\ acc\ 0_\# = acc \\
&sumTo_\#\ acc\ n \ \ = sumTo_\#\ (acc +_\#\ n)\ (n -_\#\ 1_\#)
\end{aligned}
$$

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 *sumTo* into $sumTo_\#$. But it cannot *guarantee* to do so, so performance-conscious programmers often program with unboxed values directly. As well as $Int_\#$, GHC provides a solid complement of other unboxed types, such as $Int_\#$, $Char_\#$, and $Double_\#$, 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:

$$
\begin{aligned}
&\textbf{data}\ Int = I_\#\ Int_\# \\
&plusInt :: Int \to Int \to Int \\
&plusInt\ (I_\#\ i_1)\ (I_\#\ i_2) = I_\#\ (i_1 +_\#\ i_2)
\end{aligned}
$$

Here *Int* is an ordinary algebraic data type, with one data constructor $I_\#$, that has one field of type $Int_\#$. The function *plusInt* simply pattern matches on its arguments, fetches their contents (i_1 and i_2, both of type $Int_\#$), adds them using $(+_\#)$, and boxes the result with $I_\#$.

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 ones, representing a non-terminating computation. For example, Haskell's *Bool* type is lifted, meaning that three *Bool*s are possible: *True*, *False*, and $\bot$. An unlifted type, on the other hand, is strict. The element $\bot$ 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 $Int_\#$ and $Double_\#$), Haskell has support for *unboxed tuples*. A normal, boxed tuple—of type, say, $(Int, Bool)$—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

$$divMod :: Int \to Int \to (\#\ Int, Int\ \#)$$

that returns two integers. A Haskell programmer might use *divMod* like this,

$$\textbf{case}\ divMod\ n\ k\ \textbf{of}\ (\#\ quot, rem\ \#) \to ...$$

using **case** to unpack the components of a tuple. However, during compilation, the unboxed tuple is erased completely. The *divMod* function is compiled to return two values, in separate registers, and the **case** statement is compiled simply to bind *quot* and *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: $(+) :: (\#\ Int, Int\ \#) \to Int$ compiles to the exact same code as $(+) :: Int \to Int \to Int$; 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 $(\#\ Int, (\#\ Float_\#, Bool\ \#)\ \#)$ is a distinct type from $(\#\ Int, Float_\#, Bool\ \#)$, 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 *bTwice* from the introduction:

$$
\begin{aligned}
&bTwice :: \forall\ a.\ Bool \to a \to (a \to a) \to a \\
&bTwice\ b\ x\ f = \textbf{case}\ b\ \textbf{of}\ True \to f\ (f\ x) \\
&\qquad\qquad\qquad\qquad\quad\ False \to x
\end{aligned}
$$

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

GHC therefore adopts the following principle:

- *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

[3] Remember, Haskell is a lazy language, so the second argument might not be evaluated.

[4] The suffix "#" does not imply any special treatment by the compiler; it is simply a naming convention that suggests to the reader that there may be some use of unboxed values going on.

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,[5] $\mathit{Bool} :: \mathit{Type}$, $\mathit{Maybe} :: \mathit{Type} \to \mathit{Type}$, and $\mathit{Maybe}\ \mathit{Bool} :: \mathit{Type}$. So it is natural to use the kind to classify types into the lifted and unlifted forms, thus $\mathit{Int}_\# :: \#$, $\mathit{Float}_\# :: \#$, where "#" is a new kind that classifies unlifted types[6].

In contrast, *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 Type*. For example, here is *bTwice* with an explicitly-specified kind:

$$\mathit{bTwice} :: \forall\ (a :: \mathit{Type}).\ \mathit{Bool} \to a \to (a \to a) \to a$$

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

3.2 Sub-kinding

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

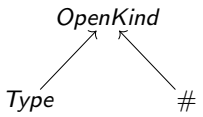
$$(\to) :: \mathit{Type} \to \mathit{Type} \to \mathit{Type}$$

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

```
instance Monad ((→) env) where
    -- note that env is the left argument to (→)
  return x      = λenv → x
  ma >>= fmb = λenv → fmb (ma env) env
```

Given the above kind for $(\to)$, the type $\mathit{Monad}\ ((\to)\ \mathit{env})$ is well-kinded. We have $(\to)\ \mathit{env} :: \mathit{Type} \to \mathit{Type}$, and that is the kind that *Monad* expects.

But now we have a serious problem: *a function over unlifted types, such as* $\mathit{sumTo}_\# :: \mathit{Int}_\# \to \mathit{Int}_\# \to \mathit{Int}_\#$*, becomes ill-kinded*! Why? Because $(\to)$ expects a *Type*, but $\mathit{Int}_\# :: \#$. 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:



That is, GHC had a kind *OpenKind*, a super-kind of both *Type* and #. We could then say that

$$(\to) :: \mathit{OpenKind} \to \mathit{OpenKind} \to \mathit{Type}$$

To avoid the inevitable complications of sub-kinding and kind inference, GHC also stipulated that only *fully-saturated* uses of $(\to)$ would have this bizarre kind; partially applied uses of $(\to)$ would get the far saner kind $\mathit{Type} \to \mathit{Type} \to \mathit{Type}$ 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 $(\to) :: \mathit{Type} \to \mathit{Type} \to \mathit{Type}$, GHC accepts types like $\mathit{Int}_\# \to \mathit{Double}_\#$.
- 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 *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 :: Int# → Int#
f n = if n <# 0# then error "Negative argument"
                 else n /# 2#
```

Here $\mathit{error} :: \forall\ a.\ \mathit{String} \to a$ 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 $\mathit{Int}_\#$. 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

$$\forall\ (a :: \mathit{OpenKind}).\ \mathit{String} \to a$$

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:

```
myError :: String → a
myError s = error ("Program error " ++ s)
```

then GHC infers the type $\forall\ (a :: \mathit{Type}).\ \mathit{String} \to a$, 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, *TYPE*

[5] The Haskell Report [9] uses the symbol "$\star$" as the kind of ordinary types, but the community seems to be coalescing around this new spelling of *Type*, which is available in GHC 8. We use *Type* rather than "$\star$" throughout this paper.

[6] Do not be distracted by the inconsistent notation here; "#" really is what GHC used in the past, but the rest of the paper shows a more uniform way forward.

[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 *Rep* as the enumeration instead of a list of unary representations.

$$TYPE :: Rep \to Type$$

with the following supporting definitions:[9]

```
type Rep       = [UnaryRep]   -- A type-level list
data UnaryRep = PtrRep        -- Boxed, lifted
              | UPtrRep       -- Boxed, unlifted
              | IntRep        -- Unboxed ints
              | FloatRep      -- Unboxed floats
              | DoubleRep     -- Unboxed doubles
              | ... etc ...
type Lifted    = '[PtrRep]
type Type      = TYPE Lifted
```

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:

```
Int          :: Type
Int          :: TYPE Lifted      -- Expanding Type
Int          :: TYPE '[PtrRep]   -- Expanding Lifted
Int#         :: TYPE '[IntRep]
Float#       :: TYPE '[FloatRep]
(Int, Bool)  :: Type
Maybe Int    :: Type
Maybe        :: Type → Type
```

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 [17]. 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:

```
(# Int, Bool #)   :: TYPE '[PtrRep, PtrRep]
(# Int#, Bool #)  :: TYPE '[IntRep, PtrRep]
(# #)             :: TYPE '[]
```

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:

```
type PFP = TYPE '[PtrRep, FloatRep, PtrRep]
(# Int, (# Float#, Bool #) #)   :: PFP
(# Int, Float#, Bool #)         :: PFP
(# (# Int, (# #) #), Float#, Bool #) :: PFP
```

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*:

```
(→) :: ∀ (r1 :: Rep) (r2 :: Rep).
       TYPE r1 → TYPE r2 → Type
error :: ∀ (r :: Rep) (a :: TYPE r). String → a
```

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 [1, 16]; we do not revisit it here.

You might also wonder whether why *TYPE* returns a *TYPE Lifted*. Why not return *TYPE* '[*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 :: ∀ (r :: Rep) (a :: TYPE r).
          Bool → a → (a → a) → a
```

Sadly, no. We cannot compile a levity-polymorphic *bTwice* into concrete machine code, because *its calling convention depends on r.*

[9] 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 type of lists containing elements of type *UnaryRep*, while '[*PtrRep*] is a list containing one element, *PtrRep*.

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 8.5 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 *levity*-polymorphic. In the implementation of *bTwice* where $a :: \mathit{Type}$, this is precisely what is done. 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 levity-polymorphic binders.* Every bound term variable in a Haskell program must have a type whose kind is fixed and free of any type variables.[10] 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 $a :: \mathit{TYPE}\ r$.
2. *Disallow levity-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 levity 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 $(\lambda x \to e)$, it must decide on a type for x. The algorithm naturally invents a unification variable[11] α. But what *kind* does α have? Equipped with levity polymorphism, GHC invents *another* unification variable $\rho :: \mathit{Rep}$ and chooses $\alpha :: \mathit{TYPE}\ \rho$. 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

$$f :: \forall\ (r :: \mathit{Rep})\ (a :: \mathit{TYPE}\ r).\ a \to a$$

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* levity polymorphism;[12] but we can for the first time *check* the declared uses of levity polymorphism. Thus, we can write

$$\mathit{myError} :: \forall\ (r :: \mathit{Rep})\ (a :: \mathit{TYPE}\ r).\ \mathit{String} \to a$$
$$\mathit{myError}\ s = \mathit{error}\ (\texttt{"Program error "} +\!\!+ s)$$

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

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

Metavariables:

x	Variables	α	Type variables
n	Integer literals	r	Representation variables

$\upsilon ::= \mathsf{P} \mid \mathsf{I}$		Concrete reps.
$\rho ::= r \mid \upsilon$		Runtime reps.
$\kappa ::= \mathit{TYPE}\ \rho$		Kinds
$B ::= \mathit{Int} \mid \mathit{Int}_\#$		Base types
$\tau ::= B \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha{:}\kappa.\ \tau \mid \forall r.\ \tau$		Types
$e ::= x \mid e_1\ e_2 \mid \lambda x{:}\tau.\ e \mid \Lambda \alpha{:}\kappa.\ e \mid e\ \tau \mid \Lambda r.\ e \mid e\ \rho \mid \mathit{I}_\#[e]$		
$\mid\ \mathbf{case}\ e_1\ \mathbf{of}\ \mathit{I}_\#[x] \to e_2 \mid n \mid \mathbf{error}$		Expressions
$v ::= \lambda x{:}\tau.\ e \mid \Lambda \alpha{:}\kappa.\ v \mid \Lambda r.\ v \mid \mathit{I}_\#[v] \mid n$		Values
$\Gamma ::= \emptyset \mid \Gamma, x{:}\tau \mid \Gamma, \alpha{:}\kappa \mid \Gamma, r$		Contexts

Figure 2. The grammar for $\mathcal{L}$

6. Correctness of levity polymorphism

We claim above (Section 5.1) that restricting the use of levity 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 levity-polymorphic language with exactly these restrictions is compilable. First, we define $\mathcal{L}$, a variant of System F [4, 14] that supports levity polymorphism. Second, we define a lower-level language $\mathcal{M}$, 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; $\mathcal{M}$ does *not* support levity polymorphism. Lastly, we define type-erasing compilation as a partial function from $\mathcal{L}$ to $\mathcal{M}$. We prove our compilation function correct via two theorems: that compilation is well-defined whenever the source $\mathcal{L}$-expression is well-typed and that the $\mathcal{M}$ operational semantics simulates that for $\mathcal{L}$.[13]

6.1 The $\mathcal{L}$ language

The grammar for $\mathcal{L}$ appears in Figure 2. Along with the usual System F constructs, it supports the base type $\mathit{Int}_\#$ with literals n; data constructor $\mathit{I}_\#$ to form *Int*; **case** expressions for unboxing integers; and **error**. Most importantly, $\mathcal{L}$ supports levity polymorphism via the novel forms $\Lambda r.\ e$ and $e\ \rho$, abstractions over and applications to

[10] Care should be taken when reading this sentence. Note that the kind polymorphism in $f :: \forall\ k\ (a :: k).\ \mathit{Proxy}\ k \to \mathit{Int}$ is just fine: the kind of f's type is *Type*! No variables there.

[11] Also called an *existential* variable in the literature. A unification variable stands for an as-yet-unknown type. In GHC, unification variables contain mutable cells that are filled with a concrete type when discovered; see [13] for example.

[12] Refusing to generalize over type variables of kind *Rep* is quite like Haskell's existing *monomorphism restriction*, where certain unconstrained type variables similarly remain ungeneralized. Both approaches imperil having principal types. In the case of levity 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 levity polymorphism.

$\boxed{\Gamma \vdash e : \tau}$ Term validity

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{E\_Var} \qquad \frac{\Gamma \vdash e : \mathit{Int}_\#}{\Gamma \vdash \mathit{I}_\#[e] : \mathit{Int}}\ \text{E\_Con}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1 \qquad \Gamma \vdash \tau_1 : \mathit{TYPE}\ \upsilon}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{E\_App}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 : \mathit{TYPE}\ \upsilon}{\Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2}\ \text{E\_Lam}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash e : \tau \qquad \Gamma \vdash \kappa\ \mathsf{kind}}{\Gamma \vdash \Lambda\alpha{:}\kappa.\ e : \forall \alpha{:}\kappa.\ \tau}\ \text{E\_TLam}$$

$$\frac{\Gamma \vdash e : \forall \alpha{:}\kappa.\ \tau_1 \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e\ \tau_2 : \tau_1[\tau_2/\alpha]}\ \text{E\_TApp}$$

$$\frac{\Gamma, r \vdash e : \tau}{\Gamma \vdash \Lambda r.\ e : \forall r.\ \tau}\ \text{E\_RLam} \qquad \frac{\Gamma \vdash e : \forall r.\ \tau}{\Gamma \vdash e\ \rho : \tau[\rho/r]}\ \text{E\_RApp}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Int} \qquad \Gamma, x{:}\mathit{Int}_\# \vdash e_2 : \tau}{\Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathit{I}_\#[x] \to e_2 : \tau}\ \text{E\_Case}$$

$$\frac{}{\Gamma \vdash \mathbf{error} : \forall r.\ \forall \alpha{:}\mathit{TYPE}\ r.\ \mathit{Int} \to \alpha}\ \text{E\_Error}$$

$$\frac{}{\Gamma \vdash n : \mathit{Int}_\#}\ \text{E\_IntLit}$$

$\boxed{\Gamma \vdash \tau : \kappa}$ Type validity

$$\frac{}{\Gamma \vdash \mathit{Int} : \mathit{TYPE}\ \mathsf{P}}\ \text{T\_Int} \qquad \frac{}{\Gamma \vdash \mathit{Int}_\# : \mathit{TYPE}\ \mathsf{I}}\ \text{T\_IntH}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1 \to \tau_2 : \mathit{TYPE}\ \mathsf{P}}\ \text{T\_Arrow}$$

$$\frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa}\ \text{T\_Var}$$

$$\frac{\Gamma, \alpha{:}\kappa_1 \vdash \tau : \kappa_2 \qquad \Gamma \vdash \kappa_1\ \mathsf{kind}}{\Gamma \vdash \forall \alpha{:}\kappa_1.\ \tau : \kappa_2}\ \text{T\_AllTy}$$

$$\frac{\Gamma, r \vdash \tau : \kappa \qquad \kappa \neq \mathit{TYPE}\ r}{\Gamma \vdash \forall r.\ \tau : \kappa}\ \text{T\_AllRep}$$

$\boxed{\Gamma \vdash \kappa\ \mathsf{kind}}$ Kind validity

$$\frac{}{\Gamma \vdash \mathit{TYPE}\ \upsilon\ \mathsf{kind}}\ \text{K\_Const} \qquad \frac{r \in \Gamma}{\Gamma \vdash \mathit{TYPE}\ r\ \mathsf{kind}}\ \text{K\_Var}$$

Figure 3. Typing judgments for $\mathcal{L}$

$$\frac{\Gamma \vdash e_2 : \tau \qquad \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{P} \qquad \Gamma \vdash e_1 \longrightarrow e_1'}{\Gamma \vdash e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{S\_AppLazy}$$

$$\frac{\Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{P}}{\Gamma \vdash (\lambda x{:}\tau.\ e_1)\ e_2 \longrightarrow e_1[e_2/x]}\ \text{S\_BetaPtr}$$

$$\frac{\Gamma \vdash e_2 : \tau \qquad \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I} \qquad \Gamma \vdash e_2 \longrightarrow e_2'}{\Gamma \vdash e_1\ e_2 \longrightarrow e_1\ e_2'}\ \text{S\_AppStrict}$$

$$\frac{\Gamma \vdash v_2 : \tau \qquad \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I} \qquad \Gamma \vdash e_1 \longrightarrow e_1'}{\Gamma \vdash e_1\ v_2 \longrightarrow e_1'\ v_2}\ \text{S\_AppStrict2}$$

$$\frac{\Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I}}{\Gamma \vdash (\lambda x{:}\tau.\ e)\ v \longrightarrow e[v/x]}\ \text{S\_BetaUnboxed}$$

$$\frac{\Gamma \vdash e \longrightarrow e'}{\Gamma \vdash e\ \tau \longrightarrow e'\ \tau}\ \text{S\_TApp} \qquad \frac{\Gamma \vdash e \longrightarrow e'}{\Gamma \vdash e\ \rho \longrightarrow e'\ \rho}\ \text{S\_RApp}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash e \longrightarrow e'}{\Gamma \vdash \Lambda\alpha{:}\kappa.\ e \longrightarrow \Lambda\alpha{:}\kappa.\ e'}\ \text{S\_TLam}$$

$$\frac{}{\Gamma \vdash (\Lambda\alpha{:}\kappa.\ v)\ \tau \longrightarrow v[\tau/\alpha]}\ \text{S\_TBeta}$$

$$\frac{\Gamma, r \vdash e \longrightarrow e'}{\Gamma \vdash \Lambda r.\ e \longrightarrow \Lambda r.\ e'}\ \text{S\_RLam}$$

$$\frac{}{\Gamma \vdash (\Lambda r.\ v)\ \rho \longrightarrow v[\rho/r]}\ \text{S\_RBeta}$$

$$\frac{\Gamma \vdash e_1 \longrightarrow e_1'}{\Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathit{I}_\#[x] \to e_2 \longrightarrow \mathbf{case}\ e_1'\ \mathbf{of}\ \mathit{I}_\#[x] \to e_2}\ \text{S\_Case}$$

$$\frac{}{\Gamma \vdash \mathbf{case}\ \mathit{I}_\#[n]\ \mathbf{of}\ \mathit{I}_\#[x] \to e_2 \longrightarrow e_2[n/x]}\ \text{S\_Match}$$

$$\frac{\Gamma \vdash e \longrightarrow e'}{\Gamma \vdash \mathit{I}_\#[e] \longrightarrow \mathit{I}_\#[e']}\ \text{S\_Con} \qquad \frac{}{\Gamma \vdash \mathbf{error} \longrightarrow \bot}\ \text{S\_Error}$$

Figure 4. Operational semantics for $\mathcal{L}$

runtime representations. Typing rules and operational semantics for $\mathcal{L}$ appear in Figure 3 and Figure 4. For the most part, these rules are straightforward. In particular, note that $\mathcal{L}$ has a stratified type system, with distinct types and kinds. While this differs from the most recent GHC, the stratification greatly simplifies this presentation; $\mathcal{L}$ still captures the essence of levity polymorphism in GHC.

The main payload of $\mathcal{L}$ is in its E_App and E_Lam rules: note the highlighted premises. We see (Figure 2) that a kind $\mathit{TYPE}\ \upsilon$ must be fully concrete, as υ stands for only P or I—never r, a representation variable. Thus rules E_App and E_Lam implement the levity-polymorphism restrictions of Section 5.1.

We wish $\mathcal{L}$ to support type erasure. For this reason, the kind (that is, runtime representation) of a type abstraction must match that of the underlying expression. We can see this in the fact that T_AllTy results in a type of kind κ_2, not $\mathit{TYPE}\ \mathsf{P}$, as one might expect in a language without type erasure. Support for type erasure is also why $\mathcal{L}$-expressions are evaluated even under Λ and why the definition for values v must be recursive under Λ. Representation abstraction, also erased, is similar to type abstraction.

Following Haskell, $\mathcal{L}$'s operational semantics supports both lazy and strict functions. The choice of evaluation strategy is type-directed.

Language $\mathcal{L}$ is type-safe:

Theorem (Preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e \longrightarrow e'$*, then* $\Gamma \vdash e' : \tau$.

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

The proofs appear in the appendix.

6.2 The $\mathcal{M}$ language

We compile $\mathcal{L}$ into $\mathcal{M}$, whose grammar appears in Figure 5 and operational semantics appears in Figure 6. The $\mathcal{M}$ language requires expressions to be in A-normal form, where a function can be called

Metavariables:

p Lifted (pointer) variables i Integer variables

$$
\begin{array}{rll}
y & ::= p \mid i & \text{Variables} \\
t & ::= t\ y \mid t\ n \mid \lambda y.t \mid y \mid \mathbf{let}\ p = t_1\ \mathbf{in}\ t_2 & \\
 & \quad \mid \mathbf{let!}\ y = t_1\ \mathbf{in}\ t_2 \mid \mathbf{case}\ t_1\ \mathbf{of}\ \mathit{I}_{\#}[y] \to t_2 \mid \mathbf{error} & \\
 & \quad \mid \mathit{I}_{\#}[y] \mid \mathit{I}_{\#}[n] \mid n & \text{Expressions} \\
w & ::= \lambda y.t \mid \mathit{I}_{\#}[n] \mid n & \text{Values} \\
S & ::= \emptyset \mid \mathsf{Force}(p), S \mid \mathsf{App}(p), S \mid \mathsf{App}(n), S & \\
 & \quad \mid \mathsf{Let}(y, t), S \mid \mathsf{Case}(y, t), S & \text{Stacks} \\
H & ::= \emptyset \mid p \mapsto t, H & \text{Heaps} \\
\mu & ::= \langle t; S; H \rangle & \text{Machine states} \\
V & ::= \emptyset \mid x \mapsto y,\ V \mid y,\ V & \text{Variable envs.}
\end{array}
$$

Figure 5. The grammar for $\mathcal{M}$

$$
\begin{array}{rcll}
\langle t\ p; S; H \rangle & \longrightarrow & \langle t; \mathsf{App}(p), S; H \rangle & \text{PAPP} \\
\langle t\ n; S; H \rangle & \longrightarrow & \langle t; \mathsf{App}(n), S; H \rangle & \text{IAPP} \\
\langle p; S; p \mapsto w, H \rangle & \longrightarrow & \langle w; S; p \mapsto w, H \rangle & \text{VAL} \\
\langle p; S; p \mapsto t, H \rangle & \longrightarrow & \langle t; \mathsf{Force}(p), S; H \rangle & \text{EVAL} \\
\langle \mathbf{let}\ p = t_1\ \mathbf{in}\ t_2; S; H \rangle & \longrightarrow & \langle t_2; S; p \mapsto t_1, H \rangle & \text{LET} \\
\langle \mathbf{let!}\ y = t_1\ \mathbf{in}\ t_2; S; H \rangle & \longrightarrow & \langle t_1; \mathsf{Let}(y, t_2), S; H \rangle & \text{SLET} \\
\langle \mathbf{case}\ t_1\ \mathbf{of}\ \mathit{I}_{\#}[y] \to t_2; S; H \rangle & \longrightarrow & \langle t_1; \mathsf{Case}(y, t_2), S; H \rangle & \text{CASE} \\
\langle \mathbf{error}; S; H \rangle & \longrightarrow & \bot & \text{ERR} \\
\langle \lambda p_1.t_1; \mathsf{App}(p_2), S; H \rangle & \longrightarrow & \langle t_1[p_2/p_1]; S; H \rangle & \text{PPOP} \\
\langle \lambda i.t_1; \mathsf{App}(n), S; H \rangle & \longrightarrow & \langle t_1[n/i]; S; H \rangle & \text{IPOP} \\
\langle w; \mathsf{Force}(p), S; H \rangle & \longrightarrow & \langle w; S; p \mapsto w, H \rangle & \text{FCE} \\
\langle n; \mathsf{Let}(i, t), S; H \rangle & \longrightarrow & \langle t[n/i]; S; H \rangle & \text{ILET} \\
\langle \mathit{I}_{\#}[n]; \mathsf{Case}(i, t), S; H \rangle & \longrightarrow & \langle t[n/i]; S; H \rangle & \text{IMAT}
\end{array}
$$

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 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 μ. 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 λ-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).

Finally, ERR processes **error** by aborting the machine.

$\boxed{[\![e]\!]^V_\Gamma \rightsquigarrow t}$ Compilation

$$\frac{x \mapsto y \in V}{[\![x]\!]^V_\Gamma \rightsquigarrow y}\ \text{C\_VAR}$$

$$\frac{\begin{array}{ll}\Gamma \vdash e_2 : \tau & \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{P} \\ p \# V & V' = V, p \\ [\![e_1]\!]^{V'}_\Gamma \rightsquigarrow t_1 & [\![e_2]\!]^{V'}_\Gamma \rightsquigarrow t_2\end{array}}{[\![e_1\ e_2]\!]^V_\Gamma \rightsquigarrow \mathbf{let}\ p = t_2\ \mathbf{in}\ t_1\ p}\ \text{C\_APPLAZY}$$

$$\frac{\begin{array}{ll}\Gamma \vdash e_2 : \tau & \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I} \\ i \# V & V' = V, i \\ [\![e_1]\!]^{V'}_\Gamma \rightsquigarrow t_1 & [\![e_2]\!]^{V'}_\Gamma \rightsquigarrow t_2\end{array}}{[\![e_1\ e_2]\!]^V_\Gamma \rightsquigarrow \mathbf{let!}\ i = t_2\ \mathbf{in}\ t_1\ i}\ \text{C\_APPINT}$$

$$\frac{\begin{array}{ll}i \# V & V' = V, i \\ \Gamma \vdash e : \mathit{Int}_{\#} & [\![e]\!]^{V'}_\Gamma \rightsquigarrow t\end{array}}{[\![\mathit{I}_{\#}[e]]\!]^V_\Gamma \rightsquigarrow \mathbf{let!}\ i = t\ \mathbf{in}\ \mathit{I}_{\#}[i]}\ \text{C\_CON}$$

$$\frac{\begin{array}{ll}p \# V & V' = V, x \mapsto p \\ \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{P} & [\![e]\!]^{V'}_{\Gamma, x:\tau} \rightsquigarrow t\end{array}}{[\![\lambda x{:}\tau.\ e]\!]^V_\Gamma \rightsquigarrow \lambda p.t}\ \text{C\_LAMPTR}$$

$$\frac{\begin{array}{ll}i \# V & V' = V, x \mapsto i \\ \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I} & [\![e]\!]^{V'}_{\Gamma, x:\tau} \rightsquigarrow t\end{array}}{[\![\lambda x{:}\tau.\ e]\!]^V_\Gamma \rightsquigarrow \lambda i.t}\ \text{C\_LAMINT}$$

$$\frac{[\![e]\!]^V_{\Gamma, \alpha:\kappa} \rightsquigarrow t}{[\![\Lambda\alpha{:}\kappa.\ e]\!]^V_\Gamma \rightsquigarrow t}\ \text{C\_TLAM} \qquad \frac{[\![e]\!]^V_\Gamma \rightsquigarrow t}{[\![e\ \tau]\!]^V_\Gamma \rightsquigarrow t}\ \text{C\_TAPP}$$

$$\frac{[\![e]\!]^V_{\Gamma, r} \rightsquigarrow t}{[\![\Lambda r.\ e]\!]^V_\Gamma \rightsquigarrow t}\ \text{C\_RLAM} \qquad \frac{[\![e]\!]^V_\Gamma \rightsquigarrow t}{[\![e\ \rho]\!]^V_\Gamma \rightsquigarrow t}\ \text{C\_RAPP}$$

$$\frac{\begin{array}{ll}[\![e_1]\!]^V_\Gamma \rightsquigarrow t_1 & i \# V \\ [\![e_2]\!]^{V, x \mapsto i}_{\Gamma, x:\mathit{Int}_{\#}} \rightsquigarrow t_2 & \end{array}}{[\![\mathbf{case}\ e_1\ \mathbf{of}\ \mathit{I}_{\#}[x] \to e_2]\!]^V_\Gamma \rightsquigarrow \mathbf{case}\ t_1\ \mathbf{of}\ \mathit{I}_{\#}[i] \to t_2}\ \text{C\_CASE}$$

$$\frac{}{[\![n]\!]^V_\Gamma \rightsquigarrow n}\ \text{C\_INTLIT} \qquad \frac{}{[\![\mathbf{error}]\!]^V_\Gamma \rightsquigarrow \mathbf{error}}\ \text{C\_ERROR}$$

Figure 7. Compilation of $\mathcal{L}$ into $\mathcal{M}$

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 $\mathcal{L}$'s operational semantics. Applications of $I_\#$ 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 $\mathcal{L}$-expression that uses levity polymorphism in a variable bound by a λ. The type system of $\mathcal{L}$ rules out this possibility. Indeed, $\mathcal{L}$'s type system guarantees that an $\mathcal{L}$-expression can be compiled:

Theorem (Compilation)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \propto V$*, then* $[\![e]\!]^V_\Gamma \rightsquigarrow t$*.*

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

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

Theorem (Simulation)**.** *Suppose* Γ *has no term variable bindings. If* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e \longrightarrow e'$*, then* $[\![e]\!]^\emptyset_\Gamma \rightsquigarrow t$*,* $[\![e']\!]^\emptyset_\Gamma \rightsquigarrow t'$*, and* $t \Leftrightarrow t'$*.*

This theorem statement requires the notion of *joinability* of $\mathcal{M}$-expressions. While the full definition appears in the appendix, intuitively, two $\mathcal{M}$-expressions t_1 and t_2 are joinable (that is, $t_1 \Leftrightarrow t_2$) 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 $\mathcal{L}$ to a substitution in $\mathcal{M}$:

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

1. $\Gamma, x{:}\tau, \Gamma' \vdash e_1 : \tau'$ *2.* $\Gamma \vdash e_2 : \tau$ *3.* $\Gamma \vdash \tau : \mathit{TYPE}\,\mathsf{P}$
4. $[\![e_1]\!]^{V, x \mapsto p, V'}_{\Gamma, x:\tau, \Gamma'} \rightsquigarrow t_1$ *5.* $[\![e_2]\!]^V_\Gamma \rightsquigarrow t_2$

Then there exists t_3 *such that* $[\![e_1[e_2/x]]\!]^{V, V'}_{\Gamma, \Gamma'} \rightsquigarrow t_3$ *and* $\mathbf{let}\ p_2 = t_2\ \mathbf{in}\ t_1[p_2/p] \Leftrightarrow t_3$*, where* p_2 *is fresh.*

This assumption is needed when considering the S_BETAPTR rule from $\mathcal{L}$'s operational semantics—we must prove that the redex and the reduct, with its substitution, compile to joinable $\mathcal{M}$-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 $\mathcal{M}$-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.[14]

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 works over substitutions of a pointer type—no levity polymorphism is to be found here.

[14] Lacking 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:

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

  However, GHC would be at a loss trying to compile $f :: F\ a \to a$, as there would not be a way to know what size register to use for the argument; the types $\mathit{Char}_\#$ and $\mathit{Int}_\#$ 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: $\mathit{Array}_\# :: \mathit{Type} \to \#$ is representative. We might imagine abstracting over a type variable $a :: \mathit{Type} \to \#$ and wish to use $\mathit{Array}_\#$ 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 $(\__\#, \__\#)\ \mathit{Bool}$ (a partially-applied unboxed tuple) can have type $\mathit{Type} \to \#$, and its calling convention bears no resemblance to that of $\mathit{Array}_\#$.

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 $\mathit{Int}_\#$ has kind $\mathit{TYPE}\ '[\mathit{IntRep}]$ while $\mathit{Char}_\#$ has kind $\mathit{TYPE}\ '[\mathit{CharRep}]$. 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:

```
($) :: ∀ (r :: Rep) (a :: Type) (b :: TYPE r).
       (a → b) → a → b
f $ x = f x
```

Note that the argument, x, must have a lifted type (of kind *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:

$$\begin{aligned}&(.) :: \forall\ (r :: \mathit{Rep})\ (a :: \mathit{Type})\ (b :: \mathit{Type})\ (c :: \mathit{TYPE}\ r).\\ &\qquad (b \to c) \to (a \to b) \to a \to c\\ &(f\ .\ g)\ x = f\ (g\ x)\end{aligned}$$

Once again, we can generalize only the return type. Unlike in the example with ($), we see that the restriction around levity-polymorphic arguments bites here: this is the only reason that we cannot generalize the kind of b. Also unlike ($), 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 *Num* class, excerpted here:

$$\begin{aligned}&\textbf{class}\ \mathit{Num}\ a\ \textbf{where}\\ &\quad (+) :: a \to a \to a\\ &\quad \mathit{abs} :: a \to a\end{aligned}$$

Haskellers use the *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 $(+_\#) :: \mathit{Int}_\# \to \mathit{Int}_\# \to \mathit{Int}_\#$ and $(+_{\#\#}) :: \mathit{Double}_\# \to \mathit{Double}_\# \to \mathit{Double}_\#$ 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 *Num* class thus:

$$\begin{aligned}&\textbf{class}\ \mathit{Num}\ (a :: \mathit{TYPE}\ r)\ \textbf{where}\\ &\quad (+) :: a \to a \to a\\ &\quad \mathit{abs} :: a \to a\end{aligned}$$

The only change is that a is no longer of type *Type*, but can have any associated *Rep*. This allows the following instance, for example:

$$\begin{aligned}&\textbf{instance}\ \mathit{Num}\ \mathit{Int}_\#\ \textbf{where}\\ &\quad (+) = (+_\#)\\ &\quad \mathit{abs}\ n \mid n <_\# 0_\# \ = \mathit{negateInt}_\#\ n\\ &\qquad\quad \mid \mathit{otherwise} = n\end{aligned}$$

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 $(+)$:

$$(+) :: \forall\ (r :: \mathit{Rep})\ (a :: \mathit{TYPE}\ r).\ \mathit{Num}\ a \Rightarrow a \to a \to a$$

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 $\mathit{Num}\ a$ constraint takes a dictionary containing the two methods that are part of our *Num* class. To be concrete, this dictionary type looks like this:

$$\begin{aligned}&\textbf{data}\ \mathit{Num}\ (a :: \mathit{TYPE}\ r)\\ &\quad = \mathit{MkNum}\ \{(+) :: a \to a \to a, \mathit{abs} :: a \to a\}\end{aligned}$$

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 $\mathit{Num}\ a$ 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 $\mathit{plusInt}_\#$ and $\mathit{absInt}_\#$. They are fully monomorphic, taking and returning $\mathit{Int}_\#$ values; there is no levity polymorphism there, so they cannot run afoul of our restrictions.

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

$$\begin{aligned}&\$d :: \mathit{Num}\ \mathit{Int}_\#\\ &\$d = \mathit{MkNum}\ \{(+) = \mathit{plusInt}_\#, \mathit{abs} = \mathit{absInt}_\#\}\end{aligned}$$

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:

$$\begin{aligned}&\mathit{abs}_1, \mathit{abs}_2 :: \forall\ (r :: \mathit{Rep})\ (a :: \mathit{TYPE}\ r).\\ &\qquad\qquad \mathit{Num}\ a \Rightarrow a \to a\\ &\mathit{abs}_1\ \ = \mathit{abs}\\ &\mathit{abs}_2\ x = \mathit{abs}\ x\end{aligned}$$

The definition for abs_1 is acceptable; there are no levity-polymorphic bound variables. However, abs_2 is rejected! It binds a levity-polymorphic variable x. And yet abs_2 is clearly just an η-expansion of abs_1. 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 *arity*, declaring how many parameters it accepts via the machine's usual convention for passing parameters to a function. The abs_1 function has an arity of 1: its one parameter is the $\mathit{Num}\ \ a$ 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, abs_2 has an arity of 2, taking also the levity-polymorphic value to operate on and returning a levity-polymorphic value. It is this higher arity that causes trouble for abs_2. When compiling, η-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

[15] We owe this use case of levity polymorphism to Baldur Blöndal, a.k.a. Iceland_jack. See `https://ghc.haskell.org/trac/ghc/ticket/12708`.

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 α has kind A, then a variable of type $\alpha*$ (that is, a pointer to α) is allowed, but a variable of type α 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++ 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:

$$\textbf{template}\langle\textbf{typename}\ T\rangle\ T\ id(T\ t)\ \{\textbf{return}\ t;\}$$

If we were now to say $id(5)$, the C++ compiler would specialize that function, producing $\textbf{int}\ id(\textbf{int}\ t)\ \{\textbf{return}\ t;\}$. Calling $id(\texttt{'x'})$ 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*.[16] Beyond those types that can be packed into an *int*, OCaml does not have first-class unboxed types.

[16] Although not necessary to support polymorphism, per se, OCaml does need to distinguish between *int*s and pointers to allow the garbage collector to do its job. Thus *int*s are all shifted left one bit and have their lowest bit set to 1, distinguishing them from word-aligned pointers.

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 monomorphization 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

[1] R. A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016.

[2] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 114–125, Paris, Sept. 1999. ACM.

[3] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247. ACM, 1993.

[4] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la theorie des types. In J. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63 – 92. Elsevier, 1971.

[5] D. Grossman. Quantified types in an imperative language. *ACM Trans. Program. Lang. Syst.*, 28(3):429–475, May 2006.

[6] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2), Mar. 1996.

[7] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, 2002.

[8] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation*. ACM, January 2001.

[9] S. Marlow (editor). Haskell 2010 language report, 2010.

[10] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml*. O'Reilly Media, 2013.

[11] M. Naftalin and P. Wadler. *Java Generics and Collections: Speed Up the Java Development Process*. O'Reilly Media, 2006.

[12] S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens. In *FPCA*, volume 523 of *LNCS*, pages 636–666, 1991.

[13] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1), Jan. 2007.

[14] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer Berlin Heidelberg, 1974. ISBN 978-3-540-06859-4. doi: 10.1007/3-540-06859-7_148. URL `http://dx.doi.org/10.1007/3-540-06859-7_148`.

[15] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76. ACM, 1989.

[16] S. Weirich, J. Hsu, and R. A. Eisenberg. System FC with explicit kind equality. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

[17] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, TLDI '12. ACM, 2012.

A. Proofs

A.1 Preliminaries

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

Definition 2 (Equivalent states)**.** *Two machine states μ_1 and μ_2 are* equivalent*, written $\mu_1 \ \mu_2$ if $gc(\mu_1) = gc(\mu_2)$.*

Definition 3 (Joinability)**.** *Two $\mathcal{M}$-expressions t_1 and t_2 are* joinable*, written $t_1 \Leftrightarrow t_2$, when there exists t_3 such that, for all S and H, there exist S_1, H_1, S_2, and H_2 such that:*

1. $\langle t_1; S; H\rangle \longrightarrow^* \langle t_3; S_1; H_1\rangle$;
2. $\langle t_2; S; H\rangle \longrightarrow^* \langle t_3; S_2; H_2\rangle$;
3. $\langle t_3; S_1; H_1\rangle \ \langle t_3; S_2; H_2\rangle$; *and*
4. $gc(\langle t_3; S_1; H_1\rangle) = \langle t_3; S', S; H'\rangle$, *where* $H \subseteq H'$.

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 $\Gamma \vdash \tau : \kappa_1$ and $\Gamma \vdash \tau : \kappa_2$, then $\kappa_1 = \kappa_2$.*

Lemma 6 (Type determinacy)**.** *If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$.*

Lemma 7 (Reduction determinacy)**.** *If $\Gamma \vdash e \longrightarrow e_1$ and $\Gamma \vdash e \longrightarrow e_2$, then $e_1 = e_2$.*

Lemma 8 (Compilation determinacy)**.** *If $[\![e]\!]^V_\Gamma \rightsquigarrow t_1$ and $[\![e]\!]^V_\Gamma \rightsquigarrow t_2$, then $t_1 = t_2$.*

Proof. Straightforward induction, appealing to Lemma 5 and Lemma 6. □

A.3 Weakening

Lemma 9 (Typing weakening)**.** *If $\Gamma_1 \vdash e : \tau$ and $\Gamma' = \Gamma_1, \Gamma_2$, then $\Gamma' \vdash e : \tau$.*

Proof. Straightforward induction on the derivation of $\Gamma_1 \vdash e : \tau$. □

Lemma 10 (Variable assignment weakening)**.** *If $[\![e]\!]^{V_1}_{\Gamma_1} \rightsquigarrow t$, $\Gamma' = \Gamma_1, \Gamma_2$, and $V' = V_1, V_2$, then $[\![e]\!]^{V'}_{\Gamma'} \rightsquigarrow t$.*

Proof. Straightforward induction on the derivation of $[\![e]\!]^{V_1}_{\Gamma} \rightsquigarrow t$, appealing to Lemma 9. □

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 $\Gamma, r, \Gamma' \vdash \kappa$ kind and $fv(\rho) \subseteq \Gamma$, then $\Gamma, \Gamma'[\rho/r] \vdash \kappa[\rho/r]$ kind.*

Lemma 12 (Representation substitution in types)**.** *If $\Gamma, r, \Gamma' \vdash \tau : \kappa$ and $fv(\rho) \subseteq \Gamma$, then $\Gamma, \Gamma'[\rho/r] \vdash \tau[\rho/r] : \kappa[\rho/r]$*

Lemma 13 (Representation substitution in expressions)**.** *If $\Gamma, r, \Gamma' \vdash e : \tau$ and $fv(\rho) \subseteq \Gamma$, then $\Gamma, \Gamma'[\rho/r] \vdash e[\rho/r] : \tau[\rho/r]$.*

Lemma 14 (Type substitution in types)**.** *If $\Gamma, \alpha{:}\kappa', \Gamma' \vdash \tau : \kappa$ and $\Gamma \vdash \tau' : \kappa'$, then $\Gamma, \Gamma'[\tau'/\alpha] \vdash \tau[\tau'/\alpha] : \kappa$.*

Lemma 15 (Type substitution in terms)**.** *If $\Gamma, \alpha{:}\kappa, \Gamma' \vdash e : \tau$ and $\Gamma \vdash \tau' : \kappa$, then $\Gamma, \Gamma'[\tau'/\alpha] \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$.*

Lemma 16 (Substitution)**.** *If $\Gamma, x{:}\tau', \Gamma' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma, \Gamma' \vdash e[e'/x] : \tau$.*

A.5 Preservation

Theorem 17 (Preservation)**.** *If $\Gamma \vdash e : \tau$ and $\Gamma \vdash e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$.*

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

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_APPLAZY: By the induction hypothesis.
  Case S_BETAPTR: By Lemma 16.
  Case S_APPSTRICT: By the induction hypothesis.
  Case S_APPSTRICT2: 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 S_TBETA: 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* Γ *has no term variable bindings and* $\Gamma \vdash v : \tau$.

1. *If* $\tau = \tau_1 \to \tau_2$*, then* $v = \lambda x{:}\tau_1.\ e$.
2. *If* $\tau = \forall \alpha{:}\kappa.\ \tau_0$*, then* $v = \Lambda \alpha{:}\kappa.\ v_0$.
3. *If* $\tau = \forall r.\ \tau_0$*, then* $v = \Lambda r.\ v_0$.
4. *If* $\tau = \mathsf{Int}$*, then* $v = \mathsf{I}_\#[n]$.
5. *If* $\tau = \mathsf{Int}_\#$*, then* $v = n$.
6. τ *is not a type variable.*

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

Theorem 19 (Progress)**.** *Suppose* Γ *has no term variable bindings. If* $\Gamma \vdash e : \tau$*, then either* $\Gamma \vdash e \longrightarrow e'$ *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:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1 \qquad \Gamma \vdash \tau_1 : \mathsf{TYPE}\ \upsilon}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{E\_APP}$$

  We have three cases, depending on the value of υ.
  Case $\upsilon = \mathsf{P}$**:** Use the induction hypothesis on e_1, yielding two cases:
    Case $\Gamma \vdash e_1 \longrightarrow e_1'$**:** We are done by S_APPLAZY.
    Case e_1 **is a value:** Lemma 18 tells us that $e_1 = \lambda x{:}\tau_1.\ e_0$. We are done by S_BETAPTR.
  Case $\upsilon = \mathsf{I}$**:** Use the induction hypothesis on e_2, yielding two cases:
    Case $\Gamma \vdash e_2 \longrightarrow e_2'$**:** We are done by S_APPSTRICT.
    Case e_2 **is a value:** Now we use the induction hypothesis on e_1, yielding two cases:
      Case $\Gamma \vdash e_1 \longrightarrow e_1'$**:** We are done by S_APPSTRICT2.
      Case e_1 **is a value:** Lemma 18 tells us that $e_1 = \lambda x{:}\tau_1.\ e_0$ and thus we are done by S_BETAUNBOXED.
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 $\Gamma \vdash e_0 \longrightarrow e_0'$**:** 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 $\Gamma \vdash e_0 \longrightarrow e_0'$**:** We are done by S_TAPP.
  Case e_0 **is a value:** Lemma 18 tells us that $e_0 = \Lambda \alpha{:}\kappa.\ v_0$ 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_TAPP, using S_RAPP and S_RBETA.
Case E_CON: We learn that $e = \mathsf{I}_\#[e_0]$. Using the induction hypothesis on e_0 gives us two possibilities:
  Case $\Gamma \vdash e_0 \longrightarrow e_0'$**:** We are done by S_CON.
  Case e_0 **is a value:** Then e is a value.
Case E_CASE: We learn that $e = \mathbf{case}\ e_1\ \mathbf{of}\ \mathsf{I}_\#[x] \to e_2$. Using the induction hypothesis on e_1 gives us two possibilities:
  Case $\Gamma \vdash e_1 \longrightarrow e_1'$**:** We are done by S_CASE.
  Case e_1 **is a value:** We see that $\Gamma \vdash e_1 : \mathsf{Int}$, and then Lemma 18 tells us that $e_1 = \mathsf{I}_\#[n]$. 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

$\boxed{\Gamma \propto V}$ Compatibility

$$\frac{}{\emptyset \propto \emptyset}\ \textsc{Compat\_Empty}$$

$$\frac{\Gamma \propto V}{\Gamma \propto V, y}\ \textsc{Compat\_Fresh}$$

$$\frac{\Gamma \propto V}{\Gamma, \alpha{:}\kappa \propto V}\ \textsc{Compat\_TyVar}$$

$$\frac{}{\Gamma, r \propto V}\ \textsc{Compat\_RepVar}$$

$$\frac{\Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{P} \qquad \Gamma \propto V}{\Gamma, x{:}\tau \propto V, x \mapsto p}\ \textsc{Compat\_P}$$

$$\frac{\Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I} \qquad \Gamma \propto V}{\Gamma, x{:}\tau \propto V, x \mapsto i}\ \textsc{Compat\_I}$$

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 $[\![e]\!]^V_\Gamma \rightsquigarrow t$.*

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

Case E_VAR: By Lemma 20 and C_VAR.
Case E_APP:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1 \qquad \Gamma \vdash \tau_1 : \mathit{TYPE}\ \upsilon}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{E\_App}$$

Inversion tells us that $\Gamma \vdash e_2 : \tau_1$ and that $\Gamma \vdash \tau_1 : \mathit{TYPE}\ \upsilon$. We now have two cases, depending on the value of υ:
Case $\upsilon = \mathsf{P}$**:** Proceed by C_APPLAZY. Choose p such that $p \# V$. We see that $[\![e_1]\!]^{V,p}_\Gamma \rightsquigarrow t_1$ and $[\![e_2]\!]^{V,p}_\Gamma \rightsquigarrow t_2$ by induction and Lemma 10.
Case $\upsilon = \mathsf{I}$**:** Similar, by C_APPINT.
Case E_LAM:

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 : \mathit{TYPE}\ \upsilon}{\Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2}\ \text{E\_Lam}$$

We have two cases, depending on the value of υ.
Case $\upsilon = \mathsf{P}$**:** By C_LAMPTR and the induction hypothesis.
Case $\upsilon = \mathsf{I}$**:** By C_LAMINT 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:

$$\frac{\Gamma \vdash e : \mathit{Int}_\#}{\Gamma \vdash \mathit{I}_\#[e] : \mathit{Int}}\ \text{E\_Con}$$

Inversion tells us that $\Gamma \vdash e : \mathit{Int}_\#$. We can thus proceed by C_CON, Lemma 10, and the induction hypothesis.
Case E_CASE:

$$\frac{\Gamma \vdash e_1 : \mathit{Int} \qquad \Gamma, x{:}\mathit{Int}_\# \vdash e_2 : \tau}{\Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathit{I}_\#[x] \to e_2 : \tau}\ \text{E\_Case}$$

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. $\Gamma, x{:}\tau, \Gamma' \vdash e_1 : \tau'$
2. $\Gamma \vdash e_2 : \tau$
3. $\Gamma \vdash \tau : \mathit{TYPE}\,\mathsf{P}$
4. $[\![e_1]\!]^{V, x \mapsto_p, V'}_{\Gamma, x:\tau, \Gamma'} \rightsquigarrow t_1$, *and*
5. $[\![e_2]\!]^{V}_{\Gamma} \rightsquigarrow t_2$

Then there exists t_3 such that $[\![e_1[e_2/x]]\!]^{V, V'}_{\Gamma, \Gamma'} \rightsquigarrow t_3$ and $\mathbf{let}\ p_2 = t_2\ \mathbf{in}\ t_1[p_2/p] \Leftrightarrow t_3$, where p_2 is fresh.

See Section 6.4 for why this is an assumption.

Lemma 23 ($\mathit{Int}_\#$ substitution/compilation)**.** *If $[\![e]\!]^{V, x \mapsto i, V'}_{\Gamma, x:\mathit{Int}_\#, \Gamma'} \rightsquigarrow t$ and $[\![v]\!]^{V}_{\Gamma} \rightsquigarrow n$, then $[\![e[v/x]]\!]^{V, V'}_{\Gamma, \Gamma'} \rightsquigarrow t[n/i]$.*

Proof. By induction on the derivation of $[\![e]\!]^{V, x \mapsto i, V'}_{\Gamma, x:\mathit{Int}_\#, \Gamma} \rightsquigarrow t$.

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 $[\![v]\!]^{V}_{\Gamma} \rightsquigarrow n$ and Lemma 10.
Case $e \neq x$**:** The substitutions are no-ops, and so we are done.
Other rules: By induction (and Lemma 16, in cases with typing rule premises).

□

Lemma 24 (Type substitution/compilation)**.** *If $[\![e]\!]^{V}_{\Gamma, \alpha:\kappa, \Gamma'} \rightsquigarrow t$, then $[\![e[\tau/\alpha]]\!]^{V}_{\Gamma, \Gamma'} \rightsquigarrow t$.*

Proof. By induction on the derivation of $[\![e]\!]^{V}_{\Gamma, \alpha:\kappa, \Gamma'} \rightsquigarrow t$, appealing to Lemma 14 and Lemma 15 in the cases with typing premises. □

Lemma 25 (Representation substitution/compilation)**.** *If $[\![e]\!]^{V}_{\Gamma, r, \Gamma'} \rightsquigarrow t$, then $[\![e[\rho/r]]\!]^{V}_{\Gamma, \Gamma'} \rightsquigarrow t$.*

Proof. By induction on the derivation of $[\![e]\!]^{V}_{\Gamma, r, \Gamma'} \rightsquigarrow t$, appealing to Lemma 12 and Lemma 13 in the cases with typing premises. □

Lemma 26 (Canonical runtime forms)**.** *Let Γ be a typing context with no bound term variables, and suppose $\Gamma \vdash v : \tau$ and $\Gamma \vdash \tau : \mathit{TYPE}\,\mathsf{I}$. Then $[\![v]\!]^{\emptyset}_{\Gamma} \rightsquigarrow n$ for some n.*

Proof. Proceed by induction on the derivation of $\Gamma \vdash \tau : \mathit{TYPE}\,\mathsf{I}$.

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 $\tau = \forall \alpha{:}\kappa.\, \tau_0$. Lemma 18 tells us that $v = \Lambda \alpha{:}\kappa.\, v_0$. Inversion then tells us that $\Gamma, \alpha{:}\kappa \vdash v_0 : \tau_0$ and that $\Gamma, \alpha{:}\kappa \vdash \tau_0 : \mathit{TYPE}\,\mathsf{I}$. The induction hypothesis tells us that $[\![v_0]\!]^{\emptyset}_{\Gamma, \alpha:\kappa} \rightsquigarrow n$. We are done by C_TLAM.
Case T_ALLREP: Similar to previous case, using C_RLAM.

□

Theorem 27 (Simulation)**.** *Suppose Γ has no term variable bindings. If $\Gamma \vdash e : \tau$ and $\Gamma \vdash e \longrightarrow e'$, then $[\![e]\!]^{\emptyset}_{\Gamma} \rightsquigarrow t$, $[\![e']\!]^{\emptyset}_{\Gamma} \rightsquigarrow t'$, and $t \Leftrightarrow t'$.*

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 ts 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:

$$\frac{\Gamma \vdash e_2 : \tau \qquad \Gamma \vdash \tau : \mathit{TYPE}\,\mathsf{P} \qquad \Gamma \vdash e_1 \longrightarrow e_1'}{\Gamma \vdash e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{S\_APPLAZY}$$

The induction hypothesis tells us that $[\![e_1']\!]^{p}_{\Gamma} \rightsquigarrow t_1'$ and that t_1 and t_1' have a common reduct t_1''. The output of the C_APPLAZY rule is $\mathbf{let}\ p = t_2\ \mathbf{in}\ t_1\ p$. We can see that

$$\begin{aligned}\langle \mathbf{let}\ p = t_2\ \mathbf{in}\ t_1\ p; S; H\rangle &\longrightarrow \langle t_1\ p; S; p \mapsto t_2, H\rangle \\ &\longrightarrow \langle t_1; \mathsf{App}(p), S; p \mapsto t_2, H\rangle \\ &\longrightarrow^* \langle t_1''; \mathsf{App}(p), S', S; p \mapsto t_2, H'\rangle\end{aligned}$$

We also see that $[\![e_1'\ e_2]\!]^{\emptyset}_{\Gamma} \rightsquigarrow \mathbf{let}\ p = t_2\ \mathbf{in}\ t_1'\ p$ and that

$$\begin{aligned}\langle \mathbf{let}\ p = t_2\ \mathbf{in}\ t_1'\ p; S; H\rangle &\longrightarrow \langle t_1'\ p; S; p \mapsto t_2, H\rangle \\ &\longrightarrow \langle t_1'; \mathsf{App}(p), S; p \mapsto t_2, H\rangle \\ &\longrightarrow^* \langle t_1''; \mathsf{App}(p), S', S; p \mapsto t_2, H'\rangle\end{aligned}$$

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:

$$\frac{\Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{P}}{\Gamma \vdash (\lambda x{:}\tau.\ e_1)\ e_2 \longrightarrow e_1[e_2/x]}\ \text{S\_BETAPTR}$$

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 $[\![e_1']\!]^{x\mapsto p_2}_{\Gamma,x:\tau_1} \rightsquigarrow t_1'$ by Theorem 21. We can see that

$$\begin{aligned}\langle \mathbf{let}\ p = t_2\ \mathbf{in}\ (\lambda p_2.t_1')\ p; S; H\rangle &\longrightarrow \langle (\lambda p_2.t_1')\ p; S; p \mapsto t_2, H\rangle \\ &\longrightarrow \langle \lambda p_2.t_1'; \mathsf{App}(p), S; p \mapsto t_2, H\rangle \\ &\longrightarrow \langle t_1'[p/p_2]; S; p \mapsto t_2, H\rangle\end{aligned}$$

The β-reduct is $e_1'[e_2/x]$. Assumption 22 gives us t_3 such that $[\![e_1'[e_2/x]]\!]^{\emptyset}_{\Gamma} \rightsquigarrow t_3$ and $\mathbf{let}\ p_3 = t_2\ \mathbf{in}\ t_1'[p_3/p_2] \Leftrightarrow t_3$. We can see that

$$\begin{aligned}\langle \mathbf{let}\ p_3 = t_2\ \mathbf{in}\ t_1'[p_3/p_2]; S; H\rangle &\longrightarrow \langle t_1'[p_3/p_2]; S; p_3 \mapsto t_2, H\rangle \\ &\longrightarrow \langle t_1'[p_3/p_2]; S; p_3 \mapsto t_2, H\rangle\end{aligned}$$

This final machine state is α-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 β-reduct.

Case S_APPSTRICT:

$$\frac{\begin{array}{ll}\Gamma \vdash e_2 : \tau & \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I} \\ \Gamma \vdash e_2 \longrightarrow e_2' & \end{array}}{\Gamma \vdash e_1\ e_2 \longrightarrow e_1\ e_2'}\ \text{S\_APPSTRICT}$$

The induction hypothesis tells us that $[\![e_2']\!]^{i}_{\Gamma} \rightsquigarrow t_2'$ and that t_2 and t_2' have a common reduct t_2''. The output of C_APPINT is $\mathbf{let!}\ i = t_2\ \mathbf{in}\ t_1\ i$. We can see that

$$\langle \mathbf{let!}\ i = t_2\ \mathbf{in}\ t_1\ i; S; H\rangle \longrightarrow \langle t_2; \mathsf{Let}(i, t_1\ i), S; H\rangle$$

Compiling the reduct $e_1\ e_2'$ yields (using Theorem 17 to show that e_2' has the same type as e_2) $\mathbf{let!}\ i = t_2'\ \mathbf{in}\ t_1\ i$. We can see that

$$\langle \mathbf{let!}\ i = t_2'\ \mathbf{in}\ t_1\ i; S; H\rangle \longrightarrow \langle t_2'; \mathsf{Let}(i, t_1\ i), S; H\rangle$$

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

Case S_APPSTRICT2:

$$\frac{\begin{array}{ll}\Gamma \vdash v_2 : \tau & \Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I} \\ \Gamma \vdash e_1 \longrightarrow e_1' & \end{array}}{\Gamma \vdash e_1\ v_2 \longrightarrow e_1'\ v_2}\ \text{S\_APPSTRICT2}$$

Lemma 26 tells us that $[\![v_2]\!]^{\emptyset}_{\Gamma} \rightsquigarrow n_2$. Compiling the redex (by C_APPINT) yields $\mathbf{let!}\ i = n_2\ \mathbf{in}\ t_1\ i$. We can see that

$$\begin{aligned}\langle \mathbf{let!}\ i = n_2\ \mathbf{in}\ t_1\ i; S; H\rangle &\longrightarrow \langle n_2; \mathsf{Let}(i, t_1\ i), S; H\rangle \\ &\longrightarrow \langle (t_1\ i)[n_2/i]; S; H\rangle \\ &= \langle t_1\ n_2; S; H\rangle \\ &\longrightarrow \langle t_1; \mathsf{App}(n_2), S; H\rangle\end{aligned}$$

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

Case S_BETAUNBOXED:

$$\frac{\Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I}}{\Gamma \vdash (\lambda x{:}\tau.\ e)\ v \longrightarrow e[v/x]}\ \text{S\_BETAUNBOXED}$$

Here, we learn that e_1 is $\lambda x{:}\tau.\ e_1'$, $e_2 = v$, $\Gamma \vdash v : \tau$, and $\Gamma \vdash \tau : \mathit{TYPE}\ \mathsf{I}$. Thus, Lemma 26 tells us that $[\![v]\!]^{\emptyset}_{\Gamma} \rightsquigarrow n$ for some n. Also, t_1, the compilation of e_1, must be by C_LAMINT and must equal $\lambda i.t_1'$ where we know $[\![e_1']\!]^{x\mapsto i_2}_{\Gamma,x:\tau} \rightsquigarrow t_1'$ by Theorem 21. We can see that

$$\begin{aligned}\langle \mathbf{let!}\ i = n\ \mathbf{in}\ (\lambda i_2.t_1')\ i; S; H\rangle &\longrightarrow \langle n; \mathsf{Let}(i, (\lambda i_2.t_1')\ i), S; H\rangle \\ &\longrightarrow \langle ((\lambda i_2.t_1')\ i)[n/i]; S; H\rangle \\ &= \langle (\lambda i_2.t_1')\ n; S; H\rangle \\ &\longrightarrow \langle \lambda i_2.t_1'; \mathsf{App}(n), S; H\rangle \\ &\longrightarrow \langle t_1'[n/i_2]; S; H\rangle\end{aligned}$$

The β-reduct is $e_1'[v/x]$. Lemma 23 tells us that $[\![e_1'[v/x]]\!]^{\emptyset}_{\Gamma} \rightsquigarrow t_1'[n/i_2]$, which is the same as the final term in the derivation above. We are done with this case.

Case E_LAM: Impossible, as λ-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 $\Gamma \vdash e_1 \longrightarrow e_1'$. 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 $[\![e_1]\!]^{\emptyset}_{\Gamma} \rightsquigarrow t_1$ and therefore that $[\![(\Lambda\alpha{:}\kappa.\ e_1)\ \tau]\!]^{\emptyset}_{\Gamma} \rightsquigarrow t_1$. The β-reduct is $e_1[\tau/\alpha]$. Lemma 24 tells us that $[\![e_1[\tau/\alpha]]\!]^{\emptyset}_{\Gamma} \rightsquigarrow t_1$, 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 $e = I_{\#}[e_0]$ and that e has stepped via S_CON; thus $\Gamma \vdash e_0 \longrightarrow e_0'$. The redex compiles via C_CON to $\mathbf{let!}\ i = t_0\ \mathbf{in}\ I_{\#}[i]$. We can see that

$$\langle \mathbf{let!}\ i = t_0\ \mathbf{in}\ I_{\#}[i]; S; H\rangle \longrightarrow \langle t_0; \mathsf{Let}(i, I_{\#}[i]), S; H\rangle$$

The reduct similarly compiles to $\mathbf{let!}\ i = t_0'\ \mathbf{in}\ I_{\#}[i]$, and we see that

$$\langle \mathbf{let!}\ i = t_0'\ \mathbf{in}\ I_{\#}[i]; S; H\rangle \longrightarrow \langle t_0'; \mathsf{Let}(i, I_{\#}[i]), S; H\rangle$$

We are thus done by the induction hypothesis, relating t_0 and t_0'.

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

Case S_CASE: Here, we know $\Gamma \vdash e_1 \longrightarrow e_1'$. The redex compiles via C_CASE to $\mathbf{case}\ t_1\ \mathbf{of}\ I_{\#}[i] \to t_2$. We can see that

$$\langle \mathbf{case}\ t_1\ \mathbf{of}\ I_{\#}[i] \to t_2; S; H\rangle \longrightarrow \langle t_1; \mathsf{Case}(i, t_2), S; H\rangle$$

The reduct compiles also via C_CASE to $\mathbf{case}\ t_1'\ \mathbf{of}\ I_{\#}[i] \to t_2$. We can see that

$$\langle \mathbf{case}\ t_1'\ \mathbf{of}\ I_{\#}[i] \to t_2; S; H\rangle \longrightarrow \langle t_1'; \mathsf{Case}(i, t_2), S; H\rangle$$

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

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

$$\begin{aligned}\langle \mathbf{case}\ I_{\#}[n]\ \mathbf{of}\ I_{\#}[i] \to t_2; S; H\rangle &\longrightarrow \langle I_{\#}[n]; \mathsf{Case}(i, t_2), S; H\rangle \\ &\longrightarrow \langle t_2[n/i]; S; H\rangle\end{aligned}$$

The reduct is $e_2[n/x]$. By Lemma 23, we see that $[\![e_2[n/x]]\!]^{\emptyset}_{\Gamma} \rightsquigarrow t_2[n/i]$, and we are done.

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

Case E_INTLIT: Impossible.

□