Levity Polymorphism

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 linchpins of modern typed programming, but it comes with a real performance penalty. We describe this penalty; offer a principled way to reason about it (kinds as calling conventions); and propose *levity polymorphism*. This new form of polymorphism allows abstractions over calling conventions; we detail and verify restrictions that are necessary in order to compile levity-polymorphic functions. Levity polymorphism has created new opportunities in Haskell, including the ability to generalize nearly half of the type classes in GHC's standard library.

CCS Concepts • **Software and its engineering** → **Polymorphism**; *Compilers*; Functional languages

Keywords unboxed types, compilation, polymorphism

1. The Cost of Polymorphism

Consider the following Haskell function:

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

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. For example, the Glasgow Haskell Compiler (GHC), a state of the art optimizing compiler for Haskell, has supported unboxed values for decades.

But unboxed values come at the price of convenience and re-use (Section 3). In this paper we describe an elegant new approach to reconciling high performance code with pervasive polymorphism. Our contributions are these:

- We present (Section 4) a principled way to reason about compiling polymorphic functions and datatypes. Types are categorized into kinds, where each kind describes the memory layout of its types. Thus the kind determines the calling convention of functions over its 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 levity polymorphism is permissible. We give a formal proof that our rules guarantee that levity-polymorphic functions can indeed be compiled into concrete code (Section 6). Along the way, we were surprised to be unable to find any prior literature proving that the widespread technique of compiling via A-normal form [7] is semantics-preserving. A key part of that proof (unrelated to levity polymorphism) is challenging, and we leave it as an open problem (Section 6.4).
- With levity polymorphism in hand, new possibilities open up—including the ability to write an informative kind for (→) and to overload operations over both boxed and unboxed types. We explore these in Section 7.

We are not the first to use kinds in this way (see Section 9), but we take the idea much further than any other compiler we know, with happy consequences. Levity polymorphism does not make code go faster; rather, it makes highly-performant code more convenient to write and more re-usable. Levity polymorphism is implemented in GHC 8.0.1.

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

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062357

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 9.

2.1 Unboxed Values

Consider this loop, which computes the sum of the integers $1 \mathinner{..} n$:

```
sumTo :: Int → Int → Int
sumTo acc 0 = acc
sumTo acc n = sumTo (acc + n) (n − 1)
```

GHC represents 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 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. For this loop, on a typical machine 10,000,000 iterations executes in less than 0.01s when using unboxed *Int*s, but takes more 2s when using boxed integers.

For performance-critical code, GHC therefore allows the programmer to write code over explicitly-unboxed values [20]. For example, GHC provides a built-in data type $Int_{\#}$ of unboxed integers [20], represented not by a pointer but by the integer itself. Now we can rewrite *sumTo* like this[4]

```
sumTo# :: Int# → Int# → Int#
sumTo# acc 0# = acc
sumTo# acc n  = sumTo# (acc +# n) (n −# 1#)
```

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

GHC's strictness analyzer and other optimizations can often transform *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 range of other unboxed types, such as $Char_{\#}$, 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:

```
data Int = I# Int#
plusInt :: Int → Int → Int
plusInt (I# i1) (I# i2) = I# (i1 +# i2)
```

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 $I_{\#}$.

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

Figure 1. Boxity and levity, with examples

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 can only 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 a heap-allocated thunk, 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 → Int → (# Int, Int #)
```

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

[3] 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.

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

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

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: $(+) :: (\#\ \mathit{Int}, \mathit{Int}\ \#) \to \mathit{Int}$ compiles to the exact same code as $(+) :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}$; 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 $(\#\ \mathit{Int}, (\#\ \mathit{Float}_{\#}, \mathit{Bool}\ \#)\ \#)$ is a distinct type from $(\#\ \mathit{Int}, \mathit{Float}_{\#}, \mathit{Bool}\ \#)$, 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:

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

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* $x :: \mathit{Int}_{\#}$ *or* $x :: \mathit{Float}_{\#}$, *or indeed with* $x :: (\#\ \mathit{Int}, \mathit{Int}\ \#)$. Actually, *bTwice* cannot even be used on a boxed unlifted value, such as a $\mathit{ByteArray}_{\#}$. 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 auto-specialization, as we discuss in Section 9.) 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.

These same complications would arise in a strict language, where polymorphism would still have to operate over types with a shared representation. The Instantiation Principle in a strict language would replace "unlifted" with "unboxed", but much of what follows would be unchanged.

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

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

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:

$$\begin{tikzcd} & \mathit{OpenKind} & \\ \mathit{Type} \arrow[ur] & & \# \arrow[ul] \end{tikzcd}$$

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:

[5] The Haskell Report [15] 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.

- 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 kind *OpenKind* would embarrassingly appear in error messages.
- The introduction of kind polymorphism [2, 31] made this situation worse, and the subsequent introduction of kind equalities [30] made it untenable.

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] 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. [6], 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 [30, 31]; this new approach would not have been easily implementable earlier.

4.1 Runtime-representation Polymorphism

Here is the design, as implemented in GHC 8.2.1. We introduce a new, primitive type-level constant, *TYPE*

```
TYPE :: Rep -> Type
```

with the following supporting definitions:

```
data Rep  = LiftedRep         -- Boxed, lifted
          | UnliftedRep       -- Boxed, unlifted
          | IntRep            -- Unboxed ints
          | FloatRep          -- Unboxed floats
          | DoubleRep         -- Unboxed doubles
          | TupleRep [Rep]    -- Unboxed tuples
          | ...etc...
type Type = TYPE LiftedRep
```

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 LiftedRep*. It is easiest to see how these definitions work using examples:

```
Int         :: Type
Int         :: TYPE LiftedRep   -- Expanding Type
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 $r :: \mathit{Rep}$. The type *Rep* specifies how a value of that type is represented. Such representations include: a heap pointer to a lifted value (*LiftedRep*); a heap pointer to an unlifted value (*UnliftedRep*); 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 *Rep*; for example we have *DoubleRep* as well as *FloatRep*.

The type *Rep* is not magic: it is a perfectly ordinary algebraic data type, promoted to the kind level by GHC's *DataKinds* extension [31]. Similarly, *Type* is just an ordinary type synonym. 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 $\mathit{Int}_{\#}$ belongs to a different kind, using a different calling convention.

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

[7] More precisely, it throws an exception.

4.2 Representing Unboxed Tuples

Unboxed tuples pose a challenge for our approach because an unboxed tuple value is stored in multiple registers or memory locations. We thus allow the $\mathit{TupleRep}$ constructor to take a *list* of constituent Reps, indicating the representations of the components of the unboxed tuples. For example:

$$\begin{array}{ll}(\#\, \mathit{Int}, \mathit{Bool}\, \#) & :: \mathit{TYPE}\ (\mathit{TupleRep}\ '[\mathit{LiftedRep} \\ & \qquad\qquad\qquad\qquad , \mathit{LiftedRep}]) \\ (\#\, \mathit{Int}_{\#}, \mathit{Bool}\, \#) & :: \mathit{TYPE}\ (\mathit{TupleRep}\ '[\mathit{IntRep} \\ & \qquad\qquad\qquad\qquad , \mathit{LiftedRep}]) \\ (\#\, \#) & :: \mathit{TYPE}\ (\mathit{TupleRep}\ '[\,])\end{array}$$

Values of the first kind are represented by two pointer register; of the second by an integer register and a pointer register. Values of the third kind are represented by nothing at all.

This design actually allows slightly less polymorphism than we could, because the ultimate representation of unboxed tuples ignores nesting. For example, the following two types both have the same representation:

$$(\#\, \mathit{Int}, (\#\, \mathit{Bool}, \mathit{Double}\, \#)\, \#)$$
$$(\#\, (\#\, \mathit{Char}, \mathit{String}\, \#), \mathit{Int}\, \#)$$

Both are represented by three garbage-collected pointers, yet they have different kinds in our design. Accordingly, no function could be polymorphic in a variable that might take on either of those two types. We considered collapsing this structure, but we found that it was tricky to do in practice—for example, it complicated the equational theory of kinds.[8] Moreover, we had no compelling use-cases, so we adopted the simpler design described here.

4.3 Levity Polymorphism

We can now give proper types to $(\rightarrow)$ and error:

$$\begin{array}{l}(\rightarrow) :: \forall\ (\mathit{r1} :: \mathit{Rep})\ (\mathit{r2} :: \mathit{Rep}). \\ \qquad \mathit{TYPE}\ \mathit{r1} \rightarrow \mathit{TYPE}\ \mathit{r2} \rightarrow \mathit{Type} \\ \mathit{error} :: \forall\ (r :: \mathit{Rep})\ (a :: \mathit{TYPE}\ r).\ \mathit{String} \rightarrow a\end{array}$$

These types are *polymorphic* in $r :: \mathit{Rep}$. 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. But note that levity polymorphism does *not* improve the performance of any compiled code; it simply serves to make a compiled function applicable to a wider range of argument types. The compiled code, however, remains the same as it always was.

4.4 The Kind of *TYPE*

Above, we gave the type of TYPE as $\mathit{Rep} \rightarrow \mathit{Type}$. That looks suspicious because Type is short for $\mathit{TYPE}\ \mathit{LiftedRep}$, 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 $\mathit{Type} :: \mathit{Type}$ [30]. 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 [4, 30]; we do not revisit it here.

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

We choose $\mathit{TYPE} :: \mathit{Rep} \rightarrow \mathit{Type}$ 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?

$$\begin{array}{l}\mathit{bTwice} :: \forall\ (r :: \mathit{Rep})\ (a :: \mathit{TYPE}\ r). \\ \qquad\qquad \mathit{Bool} \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a\end{array}$$

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

One possibility is to generate specialized versions of bTwice, perhaps at runtime. That choice comes with significant engineering challenges, albeit less so in a JIT-based system (see Section 9.3). 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:

[8] See `https://mail.haskell.org/pipermail/ghc-devs/2017-March/014007.html`

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.[9] This rule would be violated had we implemented *bTwice* with the type as given in this section: we would have to bind a variable of type $a :: TYPE\ r$.
2. *Disallow 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[10] α. But what *kind* does α have? Equipped with levity polymorphism, GHC invents *another* unification variable $\rho :: Rep$ and chooses $\alpha :: TYPE\ \rho$. If x is used in a context expecting a lifted type, then ρ is unified with *LiftedRep*—all using GHC's existing unification machinery. In terms of GHC's implementation, this is actually a *simplification* over the previous sub-kinding story.

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

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

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 busywork in the type inference engine. So instead we *never infer* levity polymorphism;[11] but we can for the first time *check* the declared uses of levity polymorphism. Thus, we can write

```
myError :: ∀ (r :: Rep) (a :: TYPE r). String → a
myError s = error ("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*.

Metavariables:

x	Variables	α	Type variables
n	Integer literals	r	Representation variables

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

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

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

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 [8, 24] that supports levity polymorphism. Second, we define a lower-level language $\mathcal{M}$, a λ-calculus in A-normal form (ANF) [7]. 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 $\mathcal{L}$.[12]

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

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

[10] 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 [22] for example.

[11] 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.

[12] 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 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.

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 \longrightarrow e_1'}{\begin{array}{l}\Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathit{I}_\#[x] \to e_2 \longrightarrow \\ \qquad\qquad \mathbf{case}\ e_1'\ \mathbf{of}\ \mathit{I}_\#[x] \to e_2\end{array}}\ \textsc{S\_Case}$$

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

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

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

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 $\mathit{TYPE}\ v$ must be fully concrete, as v stands for only P ("pointer") or I ("integer")—never r, a representation variable. Thus rules E_APP and E_LAM implement the levity-polymorphism restrictions of Section 5.1. The distinction between P and I is also critical in choosing between lazy and strict application (S_APPLAZY and S_APPSTRICT in Figure 4).

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.

Metavariables:

$$\begin{array}{llll} p & \text{Lifted (pointer) variables} & i & \text{Integer variables} \end{array}$$

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

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

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

Figure 6. Operational semantics for $\mathcal{M}$

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 extended version of this paper [5].

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 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.

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.

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.

$\boxed{\llbracket e \rrbracket_{\Gamma}^{V} \rightsquigarrow t}$ Compilation

$$\frac{x \mapsto y \in V}{\llbracket x \rrbracket_{\Gamma}^{V} \rightsquigarrow y}\ \textsc{C\_Var}$$

$$\frac{\begin{array}{ll} \Gamma \vdash e_2 : \tau & \Gamma \vdash \tau : \mathsf{\mathit{TYPE}}\,\mathsf{P} \\ p \notin \mathit{dom}(V) & V' = V, p \\ \llbracket e_1 \rrbracket_{\Gamma}^{V'} \rightsquigarrow t_1 & \llbracket e_2 \rrbracket_{\Gamma}^{V'} \rightsquigarrow t_2 \end{array}}{\llbracket e_1\ e_2 \rrbracket_{\Gamma}^{V} \rightsquigarrow \mathbf{let}\ p = t_2\ \mathbf{in}\ t_1\ p}\ \textsc{C\_AppLazy}$$

$$\frac{\begin{array}{ll} \Gamma \vdash e_2 : \tau & \Gamma \vdash \tau : \mathsf{\mathit{TYPE}}\,\mathsf{I} \\ i \notin \mathit{dom}(V) & V' = V, i \\ \llbracket e_1 \rrbracket_{\Gamma}^{V'} \rightsquigarrow t_1 & \llbracket e_2 \rrbracket_{\Gamma}^{V'} \rightsquigarrow t_2 \end{array}}{\llbracket e_1\ e_2 \rrbracket_{\Gamma}^{V} \rightsquigarrow \mathbf{let!}\ i = t_2\ \mathbf{in}\ t_1\ i}\ \textsc{C\_AppInt}$$

$$\frac{\begin{array}{ll} i \notin \mathit{dom}(V) & V' = V, i \\ \Gamma \vdash e : \mathsf{\mathit{Int}}_{\#} & \llbracket e \rrbracket_{\Gamma}^{V'} \rightsquigarrow t \end{array}}{\llbracket \mathsf{\mathit{I}}_{\#}[e] \rrbracket_{\Gamma}^{V} \rightsquigarrow \mathbf{let!}\ i = t\ \mathbf{in}\ \mathsf{\mathit{I}}_{\#}[i]}\ \textsc{C\_Con}$$

$$\frac{\begin{array}{ll} p \notin \mathit{dom}(V) & V' = V, x \mapsto p \\ \Gamma \vdash \tau : \mathsf{\mathit{TYPE}}\,\mathsf{P} & \llbracket e \rrbracket_{\Gamma, x:\tau}^{V'} \rightsquigarrow t \end{array}}{\llbracket \lambda x{:}\tau.\ e \rrbracket_{\Gamma}^{V} \rightsquigarrow \lambda p.t}\ \textsc{C\_LamPtr}$$

$$\frac{\begin{array}{ll} i \notin \mathit{dom}(V) & V' = V, x \mapsto i \\ \Gamma \vdash \tau : \mathsf{\mathit{TYPE}}\,\mathsf{I} & \llbracket e \rrbracket_{\Gamma, x:\tau}^{V'} \rightsquigarrow t \end{array}}{\llbracket \lambda x{:}\tau.\ e \rrbracket_{\Gamma}^{V} \rightsquigarrow \lambda i.t}\ \textsc{C\_LamInt}$$

$$\frac{\llbracket e \rrbracket_{\Gamma, \alpha:\kappa}^{V} \rightsquigarrow t}{\llbracket \Lambda \alpha{:}\kappa.\ e \rrbracket_{\Gamma}^{V} \rightsquigarrow t}\ \textsc{C\_TLam} \qquad \frac{\llbracket e \rrbracket_{\Gamma}^{V} \rightsquigarrow t}{\llbracket e\ \tau \rrbracket_{\Gamma}^{V} \rightsquigarrow t}\ \textsc{C\_TApp}$$

$$\frac{\llbracket e \rrbracket_{\Gamma, r}^{V} \rightsquigarrow t}{\llbracket \Lambda r.\ e \rrbracket_{\Gamma}^{V} \rightsquigarrow t}\ \textsc{C\_RLam} \qquad \frac{\llbracket e \rrbracket_{\Gamma}^{V} \rightsquigarrow t}{\llbracket e\ \rho \rrbracket_{\Gamma}^{V} \rightsquigarrow t}\ \textsc{C\_RApp}$$

$$\frac{\begin{array}{ll} \llbracket e_1 \rrbracket_{\Gamma}^{V} \rightsquigarrow t_1 & i \notin \mathit{dom}(V) \\ \llbracket e_2 \rrbracket_{\Gamma, x:\mathsf{\mathit{Int}}_{\#}}^{V, x \mapsto i} \rightsquigarrow t_2 & \end{array}}{\llbracket \mathbf{case}\ e_1\ \mathbf{of}\ \mathsf{\mathit{I}}_{\#}[x] \to e_2 \rrbracket_{\Gamma}^{V} \rightsquigarrow \mathbf{case}\ t_1\ \mathbf{of}\ \mathsf{\mathit{I}}_{\#}[i] \to t_2}\ \textsc{C\_Case}$$

$$\frac{}{\llbracket n \rrbracket_{\Gamma}^{V} \rightsquigarrow n}\ \textsc{C\_IntLit} \qquad \frac{}{\llbracket \mathbf{error} \rrbracket_{\Gamma}^{V} \rightsquigarrow \mathbf{error}}\ \textsc{C\_Error}$$

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

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* $\llbracket e \rrbracket_{\Gamma}^{V} \rightsquigarrow t$*.*

The condition $\Gamma \propto V$ requires that V has suitable mappings for the variables bound in Γ; the full definition appears in the extended version of this paper [5].

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* $\llbracket e \rrbracket_{\Gamma}^{\emptyset} \rightsquigarrow t$*,* $\llbracket e' \rrbracket_{\Gamma}^{\emptyset} \rightsquigarrow t'$*, and* $t \Leftrightarrow t'$*.*

This theorem statement requires the notion of *joinability* of $\mathcal{M}$-expressions. While the full definition appears in the extended version, 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.* $\Gamma \vdash \tau : \mathsf{\mathit{TYPE}}\,\mathsf{P}$
4. $\llbracket e_1 \rrbracket_{\Gamma, x:\tau, \Gamma'}^{V, x \mapsto p, V'} \rightsquigarrow t_1$ *5.* $\llbracket e_2 \rrbracket_{\Gamma}^{V} \rightsquigarrow t_2$

Then there exists t_3 *such that* $\llbracket e_1[e_2/x] \rrbracket_{\Gamma, \Gamma'}^{V, V'} \rightsquigarrow t_3$ *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.[13]

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.

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.

[13] 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.

This proof shows that our restrictions are indeed sufficient to allow compilation.

7. Exploiting Levity Polymorphism

We claimed earlier that levity polymorphism makes high-performance code more convenient to write, and more reusable (by being more polymorphic than before). In this section we substantiate these claims with specific examples.

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

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

Once again, we can generalize only the return type. Unlike in the example with ($), we see that the restriction around levity-polymorphic arguments bites here: we cannot generalize the kind of b. Also unlike ($), 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 [28]. An example is the *Num* class, excerpted here:

```
class Num a where
  (+) :: a → a → a
  abs :: a → a
```

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:

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

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:

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

We can now happily write $3_{\#} + 4_{\#}$ to add machine integers.[14] 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* [10], simple records of method implementations. At runtime, any function with a $\mathit{Num}\ a$ constraint takes a dictionary (that is, record) containing the two methods that are part of our Num class. To be concrete, this dictionary type looks like this:

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

The methods that the user writes are simply record selectors. The type of the $(+)$ record selector is, as we see above, levity-polymorphic (note the $\forall\ (r :: \mathit{Rep})$). *But its implementation obeys the rules of Section 5.1*: it takes a lifted argument of type $\mathit{Num}\ a$, and returns a lifted result of type $a \to a \to a$, so all is well.

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:

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

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

So far, so good, but we are treading close to the cliff. Consider this:

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

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. Practicalities

8.1 Opportunities in GHC's Libraries

While levity polymorphism is too new a feature to be widely deployed within GHC's standard libraries—we want this feature to be more battle-tested before pervasive usage—we have generalized the type of six library functions where previous versions of GHC have used special cases in order to deal with the possibility of unlifted types[15]. Even here, though, generalizing the types was not without controversy: after introducing the new type of (\$), several users loudly (and rightly, in our opinion) complained[16] that the type of (\$), as reported in GHC's interactive environment, was far too complex. The (\$) function application operator is often an early example of a higher-order function, and a type whose definition requires an understanding of levity polymorphism is not appropriate for beginners. We thus default all type variables of kind Rep to be $\mathit{LiftedRep}$ during pretty printing, unless users specify the flag `-fprint-explicit-runtime-reps`. Given that function composition (.) was not special-cased previously, we decided not to generalize its type yet, but may do so in the future. There are no other widely-used functions that we have observed to be available for levity-generalization.

In contrast to the paucity of available levity-polymorphic functions, there is a plethora of available levity-polymorphic *classes*, along the lines of the Num class above. We have identified 34 of the 76 classes in GHC's `base` and `ghc-prim` packages (two key components of GHC's standard library) that can be levity-generalized.[17]. Future work includes taking

[14] 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`.

[15] These are error, $\mathit{errorWithoutStackTrace}$, $\bot$, $\mathit{oneShot}$, $\mathit{runRW}_{\#}$, and (\$).

[16] The thread starts here: `https://mail.haskell.org/pipermail/ghc-devs/2016-February/011268.html`.

[17] The full list of classes, along with some ideas for generalizing even more classes, is here: `https://ghc.haskell.org/trac/ghc/ticket/12708#comment:29`

advantage of these opportunities and experimenting with the results, to see if inclusion in the standard library is warranted.

8.2 Implementation

The implementation of levity polymorphism was done simultaneously with that of GHC's TypeInType extension [30]. Indeed, as levity polymorphism requires kind variables of a type other than Type (forbidden before TypeInType) and kind-level equalities requires abolishing sub-kinding, these two improvements to the compiler complement each other. (It is possible to consider implementing levity polymorphism before TypeInType, just adding the ability for kind variables of type Rep.)

Because of the significant churn caused by TypeInType, it is hard to pinpoint any effect on the compiler (e.g., length/complexity of code, efficiency) due solely to levity polymorphism. However, we can report a number of pitfalls that we ran into when realizing levity polymorphism in an industrial-strength compiler:

Managing unboxed tuples proves to be fiddly. Before levity polymorphism, an unboxed tuple data constructor took twice as many arguments as its arity. For example, $(\#\,,\,\#) :: \forall\ a\ b.\ a \to b \to (\#\, a, b\, \#)$, taking two type arguments and two value arguments. Now, it takes *three times* as many arguments as its arity: $(\#\,,\,\#) :: \forall\ r1\ r2\ (a :: \mathit{TYPE}\ r1)\ (b :: \mathit{TYPE}\ r2).\ a \to b \to (\#\, a, b\, \#)$. Despite this change, regular boxed tuples still only take twice as many arguments as their arity. This all means that, in several places in the compiler (e.g., serializing and deserializing the interface files that support separate compilation), we must carefully multiply or divide by either 2 or 3 depending on whether a tuple is boxed or unboxed.

We cannot always tell whether a type is lifted. Previously, it was dead easy to tell whether a type should be treated lazily: just check its kind. Now, however, the kind of a type might be levity-polymorphic, and it is impossible to tell whether a levity-polymorphic type is lazy or strict. Indeed, one should never ask—that is the whole point of the rules in Section 5.1! Nevertheless, there has been a steady stream of bug reports that have come in over this issue, due to the fact that unlifted types can now be abstracted over and so appear in places previously unexpected. Making this situation worse, GHC uses mutable cells to track types/kinds during type inference. The functions that check the levity of a type are pure, so they cannot look into the mutable cells; this means we must update types, replacing any filled mutable cell with its contents, before checking a type's levity. (GHC calls this process *zonking*.)

Related to this challenge is that GHC 8.2 still cannot support computation (via type families) in type representations, as the presence of type families deeply confuses the code generator, when it needs to figure out which calling convention to use at a function application. This issue is "merely engineering" to get the code generator to treat type families properly, but it appears to be a tricky problem to solve cleanly.

Transformation by η-expansion has become delicate. The optimizer sometimes chooses to η-expand a function in order to improve performance. But, with levity polymorphism, we must now be careful, as not every function can be safely η-expanded. Worse, GHC primitives absolutely *must* be η-expanded, because we have no closure available for them. For example, if $(\#\,,\,\#)$ is used unsaturated at levity-polymorphic types, we must reject the program, even though $(\#\,,\,\#)$, a function, is always lifted.

Checking for bad levity polymorphism is awkward. Ideally, the type checker would be responsible for checking to see whether the user had made use of types in a safe manner. However, we can only check for bad levity polymorphism *after* type checking is complete and we have solved for all unification variables. We thus do the levity polymorphism checks in the desugarer, a separate pass completed after type inference/checking is complete. A challenge here is that the desugarer has a harder time producing informative error messages, as it tracks code contexts much more simplistically than the type checker, which has much engineering in place behind error message generation.

Part of the challenge in levity-polymorphism checking is that, while in GHC's Core (internal, typed) language [25], a function application is easy to detect, we must report errors in terms of the sugary surface syntax, where we had to identify the range of constructs that desugar to Core function applications.

8.3 Opportunities Beyond GHC

Could levity polymorphism make its way into languages beyond Haskell? Our answer is, emphatically, *yes*. Levity polymorphism is applicable to any language with support for parametric polymorphism and unboxed types. Despite its name, levity polymorphism is not at all tied to Haskell's laziness; while laziness (in concert with polymorphism) provides extra motivation for boxing, that aspect of levity polymorphism is inessential.[18]

The main challenge, however, in implementing levity polymorphism is that it is a brand-new form of abstraction, happening in the kinds of types. It is thus most appropriate for a language already equipped with kind-polymorphism; other systems would have a much larger implementation (and syntactic) burden to overcome before introducing levity polymorphism. To our knowledge, GHC Haskell is the only language supporting both unboxed types and kind polymorphism, making levity polymorphism a natural fit. It is our hope that more languages will join us in our happy neighborhood, however. When they do, we believe our work on levity polymorphism will be of great benefit.

[18] Perhaps it is better titled *boxity polymorphism*, but that doesn't have quite the same ring.

9. Related Work

There are a number of other approaches to resolving the tension between polymorphism and performance. We review the main contenders in this section.

9.1 A Single, Uniform Representation

One approach is to represent every value uniformly, whether boxed or unboxed, lifted or unlifted. For example, OCaml uses a single machine word to represent both a pointer and an unboxed integer, distinguishing the two (for the garbage collector) by stealing flag bit from the word. This solution just does not scale. It slows arithmetic (which must account for the flag bit); it is awkward on machines where floating point operations use a different register bank;[19] and it fails altogether for types that are not word-sized, like double-precision floats, unboxed tuples, or multi-values in modern processors' SSE instruction sets [14].

Java's generics mechanism [18] is more restrictive still: it works only over boxed types; no polymorphism over unboxed types is possible.

9.2 Compile-time Monomorphization

Several languages and compilers require specializing all polymorphic functions to monomorphic variants before code generation. This group includes C++ (through its template feature), the MLton Standard ML compiler [29], and Rust. Monomorphization sidesteps many of the problems we see here: the monomorphic variants are compiled independently and may thus easily work over varying representations. *But monomorphization is a non-starter for Haskell:*

- Polymorphic recursion, which has always been part of Haskell, cannot be statically monomorphized [17].
- Higher rank types [19, 22] cannot be monomorphized at all. For example, consider

  $$f :: (\forall\, a.\, a \rightarrow a \rightarrow a) \rightarrow \mathit{Int} \rightarrow \mathit{Int}$$

  The argument to f must be polymorphic, because f may call it at many different types.

These features are not considered exotic in the Haskell ecosystem—indeed they appear in widely-used libraries such as lens[20] and the Scrap-your-Boilerplate generics library [13].

In addition, monomorphization requires that a compiler consider *all* call sites of a function, which impedes separate compilation and large-scale modularity. For example, a C++ compiler specializes template functions but then must deduplicate in the linker; and the MLton compiler requires whole-program compilation, a technique that does not scale.

9.3 Run-time Monomorphization

Some system attempt to get the best of both worlds by specializing functions where possible, still providing a run-time method for monomorphization.

The C#/.NET implementation uses a JIT compiler to specialize, compile, and then run any polymorphic functions called at runtime [12, 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.

The TIL compiler uses *intensional polymorphism* (i.e., the ability to branch on types) and aggressive optimizations to monomorphize. However, polymorphic compiled functions are allowed, where varying runtime representations can be accommodated via a runtime type check. While the TIL compiler reports impressive performance on the benchmarks tested, Tarditi et al. [26] admit that all the benchmarks were single-module programs and that polymorphism in a multi-module program might indeed pose a performance challenge.

This approach forbids the possibility of type erasure, so that the necessary type information is available at runtime, which in turn imposes a performance penalty for the runtime manipulation of type representations.

9.4 Tracking Representations in Kinds

One can view the kind system we describe in this paper as a generalization of the idea originally put forth as a part of TALT [1] and Cyclone [11]. TALT is a typed assembly language [16], with a type system where each kind describes the size of the types inhabiting that kind. To wit, the system has a kind T, which classifies all types , as well as an infinite family of kinds T_i. Types of kind T_i take i bytes in memory. The system supports type polymorphism, but no kind polymorphism. Operations that need to know the size of types (for example, indexing into an array) require that a type have a kind T_i; the i is also passed at runtime. These operations are unavailable on a type of kind T, much like our restrictions on levity-polymorphic types. The design of TALT appears to be the first place in the literature where a kind is used to describe the memory layout of its inhabiting types.

Cyclone is a type-safe C-like language that builds on the success of TALT. Grossman [9, Section 4.1] describes a kind system for Cyclone supporting two kinds, A (for "any") and B (for "boxed"). This kind system supports sub-kinding, where $B <: A$. In this system, all pointers have kind B, as does type 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, much like TALT's T. Such types can appear in pointer types:

[19] See `https://www.lexifi.com/blog/unboxed-floats-ocaml` for recent discussion of work to allow unboxed `floats`.

[20] `http://hackage.haskell.org/package/lens`

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.

We can view Cyclone's A roughly as our $\forall\ (r{::}\mathit{Rep}).\ \mathit{TYPE}\ r$. However, by being more explicit about representation quantification, we can go beyond Cyclone's type system in several ways:

- Cyclone uses sub-kinding where we use polymorphism. This leads to benefits in being able to support first-class kind equalities [30] and interacts better with type inference.
- Levity polymorphism is first-class; users can even use a runtime type check (though GHC's *Typeable* feature [23]) to determine the memory representation of an as-yet-unbound argument to a function.
- Cyclone's system supports only two kinds, whereas we present an infinite family of kinds, accurately describing unboxed tuples.
- All of these features work in concert to provide the opportunities in Section 7.

9.5 Optimization by Transformation

Instead of allowing the user to write polymorphic code over unboxed types, the user could be restricted to using only boxed types, trusting (or instructing) the compiler to optimize. GHC has supporting such optimizations like this for some time [21], but these might not apply in harder scenarios. Recent work in Scala has given users more ability to write and control these optimizations [27], reducing both the need to depend on potentially-capricious decisions made by an optimizer and the need for users to write code directly over unboxed values.

9.6 Polymorphism over Evaluation Order

Recent work by Dunfield [3] describes quantification over a function's evaluation order—that is, whether a function is call-by-value or call-by-need. Although using levity polymorphism to choose between strict and lazy application is essentially a by-product of our work (where the focus is the ability to deal with types having different representations in memory), there are clear parallels between levity quantification and evaluation-order quantification, including type-directed evaluation rules. One key difference between our system and Dunfield's is that we do not need to introduce a new quantification form, piggy-backing on kind polymorphism. Dunfield's system also does not need our restriction, as that system does not allow variation in memory representation.

10. 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] K. Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 198–212. ACM, 2003.

[2] K. Crary and S. Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 233–248. ACM, 1999.

[3] J. Dunfield. Elaborating evaluation-order polymorphism. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 256–268. ACM, 2015.

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

[5] R. A. Eisenberg and S. Peyton Jones. Levity polymorphism (extended version). Technical report, Bryn Mawr College, 2017. URL `http://cs.brynmawr.edu/~rae/papers/2017/levity/levity-extended.pdf`.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Workshop on Types in Languages Design and Implementation*. ACM, 2003.

[14] G. Mainland, S. Marlow, R. Leshchinskiy, and S. Peyton Jones. Exploiting vector instructions with generalized stream fusion. In *ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, September 2013.

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

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[17] A. Mycroft. *Polymorphic type schemes and recursive definitions*. Springer, Berlin, Heidelberg, 1984.

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

[19] M. Odersky and K. Läufer. Putting type annotations to work. In *Symposium on Principles of Programming Languages*, POPL '96. ACM, 1996.

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

[21] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. In *Science of Computer Programming*, volume 32, pages 3–47. Elsevier, October 1997.

[22] 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.

[23] S. Peyton Jones, S. Weirich, R. A. Eisenberg, and D. Vytiniotis. A reflection on types. In *A list of successes that can change the world*, LNCS. Springer, 2016. A festschrift in honor of Phil Wadler.

[24] 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.

[25] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Types in languages design and implementation*, TLDI '07. ACM, 2007.

[26] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, PLDI '96, 1996.

[27] V. Ureche, A. Biboudis, Y. Smaragdakis, and M. Odersky. Automating ad hoc data representation transformations. In *International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '15. ACM, 2015.

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

[29] S. Weeks. Whole-program compilation in MLton. Invited talk at ML Workshop, Sept. 2006.

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

[31] 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.