COMPILING A NEW KIND OF FASTER CURRY

by

SHANT HAIRAPETIAN

A THESIS

Presented to the Department of Computer and Information Science and the Graduate School of the University of Oregon

September 2021

THESIS APPROVAL PAGE

Student: Shant Hairapetian

Title: Compiling a New Kind of Faster Curry

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Zena Ariola	Chair

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded September 2021

THESIS ABSTRACT

Traditionally, programming languages based on lambda calculus such as Haskell and ML do not support functions with a number of arguments (or arity) greater than one. In these languages, currying emulates the passing of multiple arguments by accepting one argument at a time and returning a closure that accepts the next. This approach is inefficient from a performance perspective. As such, Downen et al [A faster Curry with Extensional Types[2]] proposed the addition of a primitive, extensional function type that accepts all of its arguments at once. Though this new function type enabled optimization for functions across known types, generic types could not enjoy this optimization. In their follow-up paper [Kinds are Calling Conventions[1]] Downen et al proposed a new kind of system which allows for statically tracking arity information about generic types, thereby enabling faster currying and efficient compilation for these generic types. This paper will discuss the implementation of both of these changes in GHC (Glasgow Haskell Compiler).

CURRICULUM VITAE

NAME OF AUTHOR: Shant Hairapetian

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

California State University Northridge, Northridge, CA, USA

University of Oregon, Eugene, OR, USA

DEGREES AWARDED:

Bachelor of Science, Computer Science, 2019, California State University Northridge

Master of Science, Computer Science, 2021, University of Oregon

AREAS OF SPECIAL INTEREST:

Compilers

Functional Programming

Formal Methods

ACKNOWLEDGEMENTS

I'd like to first acknowledge Paul Downen for guiding me throughout this project and always being available to answer (and re-answer) my constant barrage questions. I'd also like to thank Zach Sullivan for doing a bulk of the worker/wrapper work before my time at UO. I would like to thank Zena Ariola for keeping me on track and being ever available as a resource. I'd like to thank Simon Peyton Jones, Richard Eisenberg and Sebastian Graf for meeting with me and Paul multiple times to help solve the issues we were having with the implementation. Lastly I'd like to thank my girlfriend Celine for supporting me through this process and single-handedly packing up our apartment in preparation of a move while I was writing this thesis.

Contents

List of Figures

1 Arity

1.1 Currying

In functional programming languages, functions are just another type of data to be passed around like an integer or a string. The type of a function is denoted in Haskell with the `->` (arrow) operator. Every arrow which occurs in a type signature implies a parameter. The number of applications a function expects before evaluating to an answer is known as the function's arity. In Haskell, all functions have an arity of one. Though functions may be able to effectively receive multiple arguments, they must be passed in one at a time, in separate function applications. This process is known as currying. Let's look at an example. In Fig. 1 below, is a javascript function "add" which has an arity of two. It expects that two parameters (x and y) be passed in at once in order to return their sum.

```
function add (x, y){
  return x + y;
}
add(1,2);

(add(1))(2);
```

Figure 1: Uncurried add in javascript

The curried "add" function below in Fig. 2 however, instead of accepting both parameters at once, accepts the first, then returns a function that accepts the second.

```
function add(x) {
   return (function (y) {
       return x + y;
   });
}
(add(1))(2);
```

Figure 2: Curried add in javascript

Accordingly, the function call itself has been changed to reflect this; we call "add" with the first argument "1", this returns a function which we then apply to the second argument "2". If this seems clunky, consider Fig. 3 below which shows the same curried "add" function in Haskell:

```
add :: Int -> Int -> Int
add x y = x + y

add 1 2
```

Figure 3: Curried add in Haskell

Fortunately for developers, Haskell curries functions automatically. We merely list the parameters separated by a space (which denotes function abstraction) and the compiler does the rest. Looking at the type of "add" (`add :: Int -> Int -> Int`), we see that the function expects two integers to be applied in sequence and returns a third integer, the result of the addition. Though it may look syntactically similar to the "add" function the top of Fig. 1, under the hood, the Haskell "add" function is bound to a pair of nested lambdas (as demonstrated by "add_c" at the bottom of Fig. 1). Though this allows users to utilize currying and partial application, it is not as performant as the "add" function in Fig. 1. The Haskell "add" function involves the allocation of an additional function (the intermediate function which accepts the second parameter) and an additional function call. This is obviously less efficient. The performance hit grows even larger as we increase the number of parameters. Thus, it is desirable for the compiler to be able to detect the arity of a function and transform the call to this curried "add" function to a single call that passes both 1 and 2 at the same time without the need to allocate and call the intermediate function. This would save both time and memory space. The difficulty here is that arity is not as simple as counting the arrows in the type. The type of function "f" in Fig. 4 seems to imply that "f" accepts two integers and returns a third; which is to say that "f" has an arity of 2.

```
f :: Int -> Int -> Int
f x = let z = expensive x in \y -> z + y

g :: Int -> Int
g = f 100
```

Figure 4: An deceiving arity one function

However, after receiving the first parameter, the function “expensive” (some arbitrary expensive function) is called and its value is bound to z. The function will then return a closure that accepts the second argument. The intent of the developer is for “f” to be a function of arity 1 which accepts the first integer and returns the closure `\y -> z + y` as its answer. However, if “f” were treated as an arity 2 function–so that its body is only executed when given arguments for both ”x” and ”y”—“expensive x” would be computed on every single call to “f”. Instead, the “g” function treats “f” as a function of arity 1 and applies the first argument to “f” and returns the resulting closure. This way, no matter how many times “g” is called, “expensive x” is only computed once. This is an example of how naively counting the arrows could cause a dramatic degradation in performance, which is obviously undesirable.

1.2 Polymorphism and Arity

Another reason that counting arrows to obtain arity is problematic is polymorphism. Fig. 5 below shows an example of a polymorphic function:

```
app :: (a -> b) -> a -> b
app f x = f x
```

Figure 5: The polymorphic app function

The type of the app function above contains type variables ‘a’ and ‘b’. A type variable is a placeholder for any arbitrary type. This means that “app” accepts as its first parameter, a function `(f :  a -> b)` which in turn accepts an instance of an arbitrary type and returns another, (which confusingly, could also be the same type as `a`). As its second parameter, "app" accepts some element of type 'a' (named 'x') and then returns an element of type `b` by simply applying 'f' to 'x'. This allows us to instantiate `a` and `b` with whatever types we like (within reason). if we were to call "app" with an anonymous function of type `(Int -> Int)` and a parameter of type `Int` ,

```
app (\x -> x + 1) 1
```

`a` and `b` would be instantiated with `Int`. This call applies 1 to the increment function we've provided in-line as the first parameter and returns the value 2. In another call however we could pass as the argument `f`, a function which accepts a boolean and flips it (of type `(Bool -> Bool)`) and an argument type `Bool` for bool.

```
app (\x -> not x) True}
```

In this case, `a` and `b` would be instantiated to `Bool`.

The issue this presents is that `a` itself could be a function type, such as in this next example:

```
app (\x y -> x + y) 1
```

In the call to app, f has type `(Int -> Int -> Int)` which means that `a` is instantiated with the type `(Int -> Int)` and the application done in the body is a partial application to the function (applying only a subset of the required parameters to return a value). Meaning that the call to 'app' returns a function (bound to 'curried') which expects the second parameter 'y' of 'f'. If we then apply 2 to 'curried' (`curried 2`), we finally receive our answer (3). If we naively counted arrows to determine the arity of f, we would have erroneously assumed it to have an arity of one, however, we have shown with the "curried" function that because of polymorphism, f could in fact have a larger arity.

2 A New Function Type

The solution to the problem of determining function arity proposed by Downen et al [A Faster Curry with Extensional Types[2]] is to introduce a primitive, extensional function type (∼>) (the squiggly arrow), into the intermediate language alongside its extensional function application (which is syntactically equivalent to traditional application). For functions to be extensional, they must remain equivalent under eta-conversion (the adding or dropping of a lambda abstraction and application pair that cancel each other out). In the Fig. 6 below, loop2 is the eta-expansion of loop1 and in order for them to be eta-equivalent, loop1 and loop2 must exhibit the same behavior.

```
loop1, loop2 :: Int -> Int
loop1 = loop1
loop2 x = loop2 x  -- eta expansion

x = loop1 1
y = loop2 1
```

Figure 6: Eta-equivalence

This usually holds true for functions in lazy languages such as Haskell as by default, function calls are stored in the heap as thunks and are only evaluated as needed.

Therefore, the terms x and y at the bottom of Fig. 6 are bound to similar unevaluated thunks, both of which will enter an endless loop when forced. This, unfortunately, is not the whole story; Haskell contains functionality that allows users to force the eager evaluation of terms. One of these tools is the function seq, which takes two arguments, eagerly evaluates the first to weak-head normal form then returns the second argument. Therefore (seq loop2 True) will evaluate to true, as loop1 will be evaluated to a closure due to its single parameter, while (seq loop1 True) will loop forever. The presence of such functions means that in general, eta-equivalence does not hold for Haskell terms.

A condition for eta-equivalence is that functions must never be evaluated, only called. Therefore, the type system must reject calls to seq and other such functions when invoked upon extensional functions.

By rewriting the app example with squiggly arrows, as shown below in Fig.7, the types tell us that “f” is an arity 1 function and “app” is an arity 2 function.

```
app :: (a ~> b) ~> a ~> b
app f x = f x
```

Figure 7: Extensional app

As a result, extensional functions are not invoked until all of their arguments have been passed, and those arguments must all be passed at once (i.e. no partial applications).Naively translating our “f” function from Fig. 4 to be extensional (ie replacing all of the regular arrows with squiggly arrows), as shown in Fig. 10 below,

```
f :: Int ~> Int ~> Int
f x = \y ~> let z = expensive x in z + y
```

Figure 8: Extensional f

results in a new version of “f” that cannot be invoked until both ‘x’ and ‘y’ are passed in (making it an arity 2 function). This would mean that “expensive x” would necessarily be re-computed for every invocation. In answer to this, Downen et al [A Faster Curry with Extensional Types[2]] also introduced a way for users to denote explicit currying in the type. The bracket ({}) syntax, used in Fig. 11 below, breaks up the arity of the function and signals to the compiler that “f” is in fact an arity one function that does “real” work after the first argument is applied and should return a closure which in turn accepts the second argument.

```
f :: Int ~> {Int ~> Int}
f x = let z = expensive x in Clos \y ~> z + y
```

Figure 9: Signaling true arity with square brackets

Another restriction imposed upon extensional functions in [A Faster Curry with Extensional Types[2]] is to disallow generic types in extensional functions from being instantiated with another extensional function type, thereby sidestepping the problem presented in the "Polymorphism and Arity" section completely. Though this restriction maintains the compiler's ability to count squiggly arrows to deduce arity, it bars potential optimizations [See Calling Convention].

3 Polymorphism and Compilation

Another challenge that polymorphism presents is in compiling generic functions to machine code. The difficulty is that the shape of types at runtime may vary wildly. For instance, in Fig. 5 we specialize "app" to use Ints as well as Chars, which have different runtime representations. In many languages, ASCII characters are represented as a single byte, while integers are represented with 32 or 64 bits. Thus, generating a single piece of machine code that could handle both shapes (amongst all other types that may be applied to the function) would be a challenge. Languages such as C++ compile generic functions (called templates) by generating a different version of the function for each type with which the function is called in the codebase.

For example, if the calls to app we discuseed in the polymorphism section (shown below in Fig. 10) were written using C++ templates, the compiler would generate two app functions; one corresponding to (`app :: (Int ->`

`Int) -> Int -> Int`) and one corresponding to (`app :: (Bool -> Bool) -> Bool -> Bool`.)

```
app (\x -> x + 1) 1
app (\x -> not x) True
```

Figure 10: Calls to polymorphic app function

Though this might not seem like a dramatic cost, a “real world” codebase may have ten or even a hundred times more specialized calls which would cause a significant amount of bloat in the generated machine code. As such it would be more efficient to compile generic functions to a single block of machine code. A common solution to this, used by GHC and other implementations of polymorphic languages, is to “box” types (stick them behind a reference to a heap-allocated object) by default so that all functions merely accept and return pointers of uniform size and representation. With this change it becomes trivial to generate generic machine code (as there is only a single shape to deal with), however this layer of indirection, while more space-efficient in terms of generated code, is less performant as anytime a parameter is accessed, the function must reach into the heap, ostensibly do some work on it, and then allocate a new object in the heap to return it.

To mitigate this issue GHC provides “unboxed” types that don’t sit behind a pointer. Functions that operate over these unboxed types don’t expect a pointer to a type but rather the type itself. This fix, however, is not applicable to generic functions as the shape of the input and output types is unknown at compile time. One solution is to hide every generic type behind a pointer such that every input and output has a uniform (pointer) representation. With this approach, however, generic functions take the significant performance hit of having to reach back and forth into the heap when doing work. To solve this problem, GHC separates polymorphism of type and representation, allowing functions to be polymorphic over a type but monomorphic over its representation, meaning that the generated function can be used for any type of a specific shape. As such, this shape information must be embedded somehow in a type.

3.1 TYPE and RuntimeRep

In GHC this representation information is modeled in the RuntimeRep datatype (shown below in Fig.11):

```
data RuntimeRep = VecRep VecCount VecElem
                | TupleRep [RuntimeRep]
                | SumRep [RuntimeRep]
                | LiftedRep
                | UnliftedRep
                | IntRep
                | WordRep
                | FloatRep
                | DoubleRep
                ...
```

Figure 11: The RuntimeRep type

Every type is classified by an instance of RuntimeRep, which tells the compiler the size, representation, and location of a piece of data at runtime. Values may be stored as an immediate value (e.g. IntRep, FloatRep, WordRep etc), as a pointer to a heap-allocated object (LiftedRep or UnliftedRep), or as a combination of multiple runtime representations (TupleRep and SumRep).

In Haskell, types are further classified by kinds (the "type of a type"). An instance of RuntimeRep is embedded in the kind of a type. The kind of all types in Haskell are in the form:

```
TYPE (rr :: RuntimeRep)
```

where TYPE is a type constructor (a function that accepts a type and returns a new one) and rr is the representation of a type, i.e how members of a type should be stored and accessed in memory. If a generic type 'a' for example, has a kind of “TYPE IntRep”, we’d know that ‘a’ is represented as an immediate integer value. Or if the kind of ‘a’ was “TYPE TupleRep [WordRep, FloatRep]” we’d know that ‘a’ is represented as a combination of an ordinary data register (WordRep) and floating-point register (FloatRep).

The default kind for all Haskell terms is (TYPE LiftedRep) or * for short. Types of kind (TYPE LIftedRep) or (TYPE UnliftedRep) correspond to “boxed” values (a pointer to a heap-allocated object). If however the parameter to TYPE is IntRep, WordRep, FloatRep, or DoubleRep, etc, then that kind corresponds to an “unboxed” value which means that instead of a pointer to the object, the object itself is expected. Boxed values are then bifurcated into Lifted and Unlifted values. The “liftedness” (or levity) of a term informs whether it is always a value or might be a thunk (an unevaluated, nullary function that will return the value).

Unlifted types correspond to ordinary values such as the integers 0,1,2, etc, or pointers to arrays of those values. Lifted types however correspond to unevaluated expressions called thunks. Because these expressions are unevaluated, it’s possible that their computation may diverge and never return a value. Due to this, lifted types have an additional bottom element \|” which corresponds to such a non-terminating computation. As Haskell is lazy by default, parameters passed to functions are normally not evaluated right away but stored as thunks in the environment only to be evaluated if the term is needed in the body of the function (Call-By-Need). This means that a boxed value could be lifted (reference to a heap allocated thunk) or unlifted (a reference to a heap allocated value) however unboxed (non-referenced) values cannot be thunks and thus must be Unlifted.

3.2 Representation Polymorphism

Going beyond types, would it be possible to extend polymorphism to the representation as well? Traditionally, polymorphism refers to type polymorphism: functions that operate over all types. In the previous section, we presented a way to compile type polymorphic functions by monomorphising (forcing a concrete representation or shape) of the input and output types. This extra constraint allows the compiler to efficiently compile generic functions but would it be useful to have a function that has input or outputs that are polymorphic over the representation as well? Let's take a look at the 'error' function in Haskell whose type has historically been:

```
error :: [Char] -> a
```

What 'error' does is accept a string (an error message) and then outputs it to the screen instead of returning an output (which can be any type as it is bound to the generic 'a'). What this allows developers to do is output debug data to users without breaking typing rules. As such, 'error' has a polymorphic output to allow developers to insert it for debugging purposes into any function. The interesting thing about error however, is that it never actually returns a value of type 'a', but instead halts execution and outputs to a device. Therefore it never actually reads or writes a value of a and consequently does not need to know anything about the shape of `a`.

We will find that by adding a simple restriction we can in fact compile representation polymorphic code. If we were to maintain our monomorphic representation constraint, we would have to introduce a version of error for every possible representation despite the fact that the compiled code for all of the versions would be identical (because no data corresponding to those shapes is actually touched by the 'error' function). To solve this problem we could relax the monomorphic representation constraint in cases where the function does not read or write values of the polymorphic type. That would mean in these special cases, we can have functions which are not only polymorphic in the type but in the representation as well. As such with representation polymorphism we can define 'error' with the type:

```
error :: forall (r :: RuntimeRep) . forall (a :: TYPE r) .
         => [Char] -> a
```

Above, the type a above, has the kind 'TYPE r' where 'r' stands for all possible RuntimeRep values. Though now we can define a single 'error' function which will work for all representations, our new constraint of limiting representation polymorphic functions to functions which don't read or write polymorphic values seems to render this feature useless. One area where representation polymorphic functions shine is in typeclasses.

Typeclasses in Haskell are similar to interfaces in Java. They provide users a way to define polymorphic functions or operations (an interface) which can then be implemented for specific types. This is a different kind of polymorphism than the generic functions presented so far in the paper. In previous sections we discussed generic functions which compiled to a single block of code that handled a range of types. This is known as parametric polymorphism; type variables, or parameterised types are used to specify the behavior of a function. Typeclasses embody what's known as ad-hoc polymorphism. Similar to the polymorphism provided by C++ operator overloading, ad-hoc polymorphism involves maintaining a set of independently defined, monomorphic functions which are invoked by the runtime system based upon the usage context. Let's look at Haskell's most commonly used typeclass (shown below in Fig. 12), Num.

```
class  Num a  where
   (+) :: a -> a -> a
   (*) :: a -> a -> a
   negate :: a -> a
   ...
```

Figure 12: The Num typeclass

The reason that the Num typeclass is useful is that there are multiple distinct numerical types for which we need the addition function. It would be tedious to have to create a different addition, multiplication and negation function for floats, 32-bit integers and 64-bit integers, so instead we define the typeclass Num which ranges over all of the numerical types and allows the user to perform arithmetic operations upon all of them with a unified syntax. In order to accomplish this users must create an instance of the Num typeclass for every type which is a member of the class. The Num instance for Integer, for example, looks something like this:

```
instance  Num Integer  where
   (+) = plusInteger
   (*) = timesInteger
   negate = negateInteger
   ...
```

The code above declares that Integer is a member (or instance) of the Num typeclass and provides the appropriate integer addition, multiplication and negation functions for each. Users would also provide an instance for float, Int32 and whatever other numerical types are relevant. The compiler maintains a record of all of the members of a typeclass and their corresponding arithmetic operations and can then substitute the appropriate operation at a call site based upon the type of the arguments passed to the operation. Traditionally the kind of the type variable ‘a’ in the specification of Num has been monomorphic in the representation, meaning that all members of a typeclass must have the same representation. This creates an issue similar to the one presented with the ‘error’ function; a new class would need to be created for every representation, meaning a separate Num class for every single unboxed representation (ie IntRep, FloatRep, DoubleRep).

It turns out that it is in fact possible to utilize representation polymorphism to create a definition of Num which ranges over numerical types of any representation. The code below specifies that the kind of ‘a’ can have any representation.

```
class  Num (a :: TYPE r) where
   (+) :: a -> a -> a
   (*) :: a -> a -> a
   negate :: a -> a
   ...
```

But how can we do this if we cannot read or write representation-polymorphic data? The answer is that this generic function doesn’t do any of the real work at runtime. It instead merely cross references the types in the context with the list of records corresponding to the functions defined by the members of num and returns the appropriate, monomorphic instance definition. In the typical case where a typeclass instance is specialized to a concrete type, like Num Integer, the compiler statically knows the representation of arguments passed to those functions. As such, with this change, we can add unboxed types to the Num typeclass.

3.3 Calling Convention

So far we have established that representation information is needed for compilation, but is this enough? As mentioned in the “New Function Type” section, the new kind of primitive functions introduced by Downen et al [A Faster Curry with Extensional Types[2]] must have all of their arguments passed at once. In order to accomplish this, the arity of functions must be known statically. For concrete function types, the compiler can merely count the number of squiggly arrows and generate code which will pull the appropriate number of arguments off of the stack.

Once again, however, polymorphism rears its ugly head; this is to say that when the return or parameter type of a function is not known, or generic, the compiler cannot generate the corresponding machine code. The solution proposed in [A Faster Curry with Extensional Types[2]] is to simply disallow generics to be instantiated with extensional functions. The implication is that in the code in Fig. 13 below, the type ‘a’ in ‘app’ cannot be expanded to be of the form `(x ~> y)`, in effect constraining ‘f’ to be of arity 1.

```
app :: (a ~> b) ~> a ~> b
app f x = f x

app (\x ~> \y -> x + y)) 1 2
```

Figure 13: Extensional app with single arity add function

This means that if we modified our definition of "app" from Fig. 5 to use the squiggly arrows, the first argument to "app" in call such as (`app (y -> x + y) 1`) could not be translated to (`x ~> y ~> x + y`) as that would violate the constraint. Thus, the second arrow would need to be a regular (`->`) arrow (`x ~> y -> x + y`), barring us from passing both 'x' and 'y' at once. In [Kinds are Calling Conventions[1]], Downen et al propose that the kind of a type should encode not only its representation, but its calling convention (or arity) as well. Thus, TYPE would accept an additional parameter corresponding to the calling convention of a type. Let's modify our definition of app to include a calling convention for 'b'. In the first line of Fig. 14 below, we are signaling to the compiler that 'b' is an arity 1 function which means that 'f' is expected to be of arity 2: arity 1 from the kind signature of 'b' plus the single squiggly arrow in the signature of f.

```
app :: (b :: TYPE BoxedRep Call[1]) (a ~> b) ~> a ~> b
app f x = f x

three = app (\x ~> \y ~> x + y)) 1 2
```

Figure 14: Extensional app for arity 2 functions

And now, because f is an arity 2 function, 'app' becomes an arity 3 function: it must accept 'f' as well as its two arguments. But the definition of app only has two arguments defined ('f' and a single argument 'x') and according to the rules of extensional functions we must pass in all of the arguments at once and cannot return a partial application, so what gives? Remember that as discussed previously, extensional functions enjoy eta-equivalence meaning that 'app' can be soundly eta-expanded; so behind the scenes, the compiler will expand the definition of app to include the missing parameter (Show in Fig. 15 below).

```
app :: (b :: TYPE BoxedRep Call[1]) (a ~> b) ~> a ~> b
app f x y = f x y -- eta expansion
three = app (\x ~> \y ~> x + y)) 1 2
```

Figure 15: Eta-expanded app

Now, even though the compiler does not know the specific type of 'a' it knows its arity and can therefore generate machine code for applying f. But what would the calling convention be for terms that are not extensional functions? Regular functions (`->`) are merely pointers to heap allocated thunks so were therefore represented as the default kind (`TYPE LiftedRep`), but is a term's levity the same thing as its representation? Both terms with LiftedRep and UnliftedRep are represented by pointers to the heap: LiftedRep signifies pointers to thunks in the heap which must be evaluated to a value (lazy evaluation) and UnliftedRep signifies pointers to already evaluated objects in the heap (eagerly evaluated). Therefore the difference between LiftedRep and UnliftedRep is not one of representation, but rather, of evaluation strategy. Because we know that extensional functions are by definition never evaluated but only called, terms will either have, in the case of extensional functions, an arity, or for everything else, an evaluation strategy. Thus Downen et al [Kinds are Calling Conventions[1]] propose collapsing LiftedRep and UnliftedRep into a single representation corresponding to a pointer (BoxedRep),and adding an additional constructor (Eval) to the calling convention which accepts an evaluation strategy (Lifted or Unlifted). Therefore, the default kind changes from (`TYPE LiftedRep`) to (`TYPE BoxedRep (Eval Lifted)`).

Let us now translate our definition from Fig. 14 to this new kind representation. In Fig. 16 below, we have replaced UnliftedRep with BoxedRep in kind signature for 'b'(as Call[1] implies an extensional function and therefore unliftedness) and we have specified the default kind for 'a'; (which is redundant as (`TYPE BoxedRep (Eval Lifted)`) is the default kind).

```
app :: (b :: TYPE BoxedRep Call[1])
       (a :: TYPE BoxedRep (Eval Lifted))
       (a ~> b) ~> a ~> b
app f x = f x

three = app (\x ~> \y ~> x y)) (\x ~> x) 2
```

Figure 16: Extensional app with default `a`

Given this default kind, we expect that the value 'x' of type 'a' is represented as a pointer to a heap-allocated thunk. However, the call to app on the third line of Fig. 15 tries to pass the extensional function `(x ~> x)` for 'x' which is a kind error because `(x ~> x)` of type (`Int ~> Int`) has the convention "Call[1]" instead of "Eval Lifted" as expected by 'app's type. Can we fix this by making `a` polymorphic in the convention? Making the convention of "a" polymorphic, as it is below in Fig. 17, means that "app" cannot call 'x' because it has no way of knowing how many arguments 'x' expects at run-time. The compiler can't generate code for calling an extensional, primitive function of unknown arity.

```
app :: (b :: TYPE BoxedRep Call[1]) (a :: TYPE BoxedRep c)
       (a ~> b) ~> a ~> b
app f x = f x

three = app (\x ~> \y ~> x y)) (\x ~> x) 2
```

Figure 17: Extensional a with convention polymorphic `a`

However, if a convention-polymorphic value is not being called, but is just passed around like "x" is inside the body of "app", then the compiler doesn't need to know how to generate code for calling "x". In this case, it is permissible to allow a generic type to have a polymorphic convention, just like 'a' does in Fig. 17. Even though the convention of a is polymorphic in the definition of app, in our call to app, a is instantiated by the function `(x ~> x)` whose type will be inferred as (`Int ~> Int`) and will therefore have a concrete arity of 1. As such, at the point of its invocation, its arity is known and the function can therefore be applied.

4 Implementation

4.1 The Extensional Arrow

We first introduce the extensional function type into Core (the intermediate representation of Haskell) alongside the normal function type. As we are introducing the new function only in Core and not in the Haskell source, these extensional arrows will only be introduced after the source Haskell program has already been translated into Core; meaning that we will effectively be transforming regular functions to extensional functions in a manner transparent to the user. But can we just do this arbitrarily and replace any and all occurrences of regular functions with extensional ones?

4.2 The Worker/Wrapper Transformation

Let's look at a simple example with unboxed types. In this scenario, we want to improve the performance of the double function by modifying it to operate upon unboxed integers rather than boxed ones. In Fig. 18 "double" is the original function and "`double'`" is the naively transformed, unboxed version.

```
double :: Int -> Int
double x = x * 2


double' :: Int# -> Int  -- naive substitution
double' x = x * 2
```

Figure 18: Naive unboxing transformation

The code generator would, based upon type of double' (`Int -> Int`), generate code for manipulating a raw integer value rather than a pointer. This would save time as instead of reaching into the heap and evaluating a thunk, we can refer directly to integer values stored inside of a register (theoretically). The issue with this transformation, however, is that although the generated code for double' now expects a raw integer value, none of the call-sites have been updated to reflect this change. Thus, double' will be invoked with boxed integers and the code will treat the pointers as literal integer values. This would undoubtedly result in undesirable behavior. Now let's take a look at an example with extensional types.

The function "plus3" in Fig. 19 below, accepts an unboxed integer (x :: Int), then returns a closure that accepts a second Int.

```
plusx :: Int# -> Int# -> Int#
plusx = \x -> \y -> addInt# x y

plusx 1 1

plusx :: Int# ~> Int# ~> Int# -- naive subsitution
plusx = \x ~> \y ~> addInt# x y

plusx 1 1
```

Figure 19: Naive extensional transformation

When the second integer is applied to this new closure, a third closure will be returned which accepts the final integer argument, then calculates the sum of the arguments. Just below the definition of plus3, we have defined plus3' where we have naively substituted in extensional function arrows. The code generated for plus3', guided by its type, expects two registers with unboxed integer values which it must accept at once. Instead, only one of those registers will have an integer value and the other will likely have bits leftover from a previous call. This will result in erroneous and unexpected output. As we can see, arbitrarily changing types naively in this manner is unsound. To make this substitution safely, we must instead use the worker/wrapper transformation. This involves introducing an outer "wrapper" function that maintains the original type and acts as interface to the "worker" function which has the new modified type. Below in Fig. 20, we have the worker/wrapper transformation for the 'plusx' function.

```
plusx :: Int# -> Int# -> Int# -- wrapper
plusx = \x -> \y -> \z -> wplusx x y z

wplusx :: Int# ~> Int# ~> Int# -- worker
wplusx = \x ~> \y ~> \z ~> addInt# (addInt# x y) z
```

Figure 20: Extensional worker/wrapper transformation

The wrapper function 'plus3' accepts 3 parameters one at a time and then invokes the extensional worker function 'wplus3' once the call is saturated. The wrapper function acts as a guard against under-application as any partial applications of the functions will be translated to partial applications of the conventional wrapper function rather than the extensional worker function which must receive all its arguments at once. In this way the wrapper acts as an eta-expansion layer between applications of the original function and the new extensional workers generated by the transformation.

Through function inlining and beta reduction (as show in Fig. 21 below), the compiler can then translate the call 'plus3 1 1 1' to the more efficient call 'wplus3 1 1 1' in a type-safe manner.

```
    plusx 1 1
--> (((\x -> \y -> wplusx x y) 1 ) 1) -- inlining
--> wplusx 1 1                        -- beta reduction
```

Figure 21: Inlining and beta-reduction of worker/wrapper

We implemented this worker wrapper transformation as an optimization pass in GHC's Core-to-Core pipeline. After a source Haskell program has been typechecked, it is translated to GHC's intermediate language, Core, which closely resembles lambda calculus. Once this translation occurs, the compiler will optimize the generated Core code in a series of Core-to-Core transformations. In Core, all top level function definitions are represented as let bindings. The transformation works by inspecting all let expressions which bind functions, doing an arity analysis of the function, and then breaking up the function into a wrapper function with the original type and an extensional worker function whose type encodes the arity obtained from the analysis. The arity analysis we used is a fairly naive technique which involves counting lambdas for first order functions and for higher order functions,inspecting the call sites of functions passed in as arguments, and calculating a lower bound of each arity.In the simple (first order function) case, we can merely replace the arrows in the original function type to obtain the type of the worker as outlined in the plus3 example above. In the case of higher-order functions we must inspect the body of the function to obtain the type of the worker.

```
foo f =
 let a = f 1 2
     b = f 2
 in a + b 3
```

If we look at the higher order function above, our original inferred type would be (`(Int -> Int -> Int) -> Int`). Our arity analysis function then would inspect the body of foo and observe that f is called twice; once with a single argument, and once with two. To err on the side of caution, we assume the lower of arity to be one. Accordingly (as shown below in Fig. 22), we create a worker function, fooWorker, which accepts an arity one function (the second argument of f is applied in the traditional fashion) and a wrapper, fooWrapper, which constructs the extensional function to pass to the worker.

```
fooWrapper :: (Int -> Int -> Int) -> Int
fooWrapper f = fooWorker (\x ~> f x)

fooWorker :: (Int ~> Int -> Int) ~> Int
fooWorker f =
let a = f 1 2
    b = f 2
in a + b 3
```

Figure 22: Worker/Wrapper transformation of higher order function

4.3 RuntimeInfo and Friends

Though we have justified the existence of calling-convention as its own additional argument to the TYPE constructor, the declarations of many kind-polymorphic function types (such as the function type constructor above) become more cumbersome to write out. Instead of needing to quantify over two kind variables, we must now quantify over four (shown in Fig. 23 below).

```
TYPE :: RuntimeRep -> CallingConv -> Kind

(->) :: forall (r1 :: RuntimeRep) (c1 :: CallingConv)
               (r2 :: RuntimeRep) (c2 :: CallingConv).
       TYPE r1 c1 -> TYPE r2 c2 -> Type
```

Figure 23: TYPE as specified in *Kinds are Calling Conventions*[1]

In order to ease this burden, we have introduced the RuntimeInfo kind (shown in Fig. 24 below), which has a single constructor RInfo that takes a RuntimeRep as well as a CallingConv; and instead of having TYPE accept a RuntimeRep and a CallingConv, it now accepts a single RuntimeInfo kind.

```
data RuntimeInfo = RInfo RuntimeRep CallingConv

* = TYPE (RInfo LiftedRep ConvEval)
```

Figure 24: The RuntimeInfo kind

With this change, the default kind changes from (`TYPE LiftedRep ConvEval` to (`TYPE (RInfo LiftedRep ConvEval)`) [1]. Though this makes the default kind a bit more cumbersome, we can once again express the type of (`->`) with only two kind variables (shown below in Fig. 25).

```
TYPE :: RuntimeInfo -> Kind

(->) :: forall (r1 :: RuntimeInfo) (r2 :: RuntimeInfo).
       TYPE r1 -> TYPE r2 -> Type
```

Figure 25: The Definition of `->` with new TYPE

The problem that this creates is that in some cases, we want to access the representation or convention field of RuntimeInfo. Below in Fig. 26, we have the type constructor for unboxed tuples of size 2, which accepts two RuntimeReps and two CallingConventions (a pair for each element) and uses the RuntimeRep's to build to the representation of the resulting unboxed tuple, via the TupleRep constructor.

```
(#,#) :: forall (r1 :: RuntimeRep) (c1 :: CallingConv)
                (r2 :: RuntimeRep) (c2 :: CallingConv).
        TYPE (RInfo r1 c1)
     -> TYPE (RInfo r2 c2)
     -> TYPE ('RInfo ('TupleRep '[r1, r2]) 'ConvEval)
```

Figure 26: The type constructor for unboxed pair

Thus, to get around having to instantiate two kind variables for every element, we must have some way of extracting the fields of a RuntimeInfo type variable. Using closed type families we created the type-level functions (`GetRep`

[1]Note that in our implementation we preserved the LiftedRep constructor in RuntimeRep rather than moving levity information to the CallingConvention field. This is because this area of GHC is currently in flux. Thus, we have left the moving the levity information to future work.

`:: RuntimeInfo -> RuntimeRep`) and (`GetConv :: RuntimeInfo -> CallingConv`) (shown below in Fig. 27).

```
type family GetRep (ri :: RuntimeInfo) :: RuntimeRep where
 GetRep ('RInfo rep _) = rep

type family GetConv (ri :: RuntimeInfo) :: CallingConv where
 GetConv ('RInfo _ conv) = conv
```

Figure 27: The GetRep and GetConv type families

Then, using GetRep, we can extract the RuntimeRep from a RuntimeInfo type variable directly in the declaration of the type constructor in our previous example (show below in Fig. 28), enabling us to only quantify over a single RuntimeInfo variable per element in the unboxed tuple.

```
(#,#) :: forall (r1 :: RuntimeInfo) (r2 :: RuntimeInfo).
        TYPE r1
     -> TYPE r2
     -> TYPE ('RInfo ('TupleRep '[GetRep r1, GetRep r2]) 'ConvEval)
```

Figure 28: The unboxed pair type constructor with GetRep

4.4 Wiring in New Types

In GHC there are a set of types that are known globally within the compiler, either just by their unique identifier (known-key types), or by both their identifier and their definition (wired-in types). The unique identifiers for these types are available globally and defined in the compiler/GHC/Builtin/Names.hs module; these identifiers are then used to instantiate TyCon's (type constructors) which represent the type, and then are ultimately converted to types using the mkTyConTy function from compiler/GHC/Core/TyCo/Rep.hs. It is important to note that within the compiler, all wired-in types are of type Type and can only be differentiated by querying the unique identifier with which they were instantiated, effectively making them untyped within the compiler. To wire-in a type, its corresponding TyCon must be added to the wiredInTyCons list in compiler/GHC/Builtin/Types.hs. Wired-in types include several ubiquitous types like Int, Bool, String, and kind constructors such as TYPE and RuntimeRep. The constructors for these types are defined in GHC.Builtin.Types, not with traditional Haskell syntax, but rather, directly in Core via the GHC.Core.DataCon data type.A subset of these types, which are used to denote kinds (such as TYPE and RuntimeRep), are promoted to the kind-level using the "promoteDataCon" function from compiler/basicTypes/DataCon.hs.

4.4.1 Type Definitions

Our first order of business was to add the new types corresponding to CallingConv, RuntimeInfo, GetConv and GetRep to GHC.Builtin.Types (alongside their unique identifiers in GHC.Builtin.Names). Below in Fig. 29, we have the definition of our new CallingConv type and its Haskell source equivalent below.[2]

[2]Note: Though the Kinds Are Calling Conventions[1] paper specified that the LiftedRep and UnliftedRep constructors be extracted to a field in the Eval constructor in the new CallingConvention type, we have left them in RuntimeRep. This area of the GHC codebase is currently under flux (see GHC Proposal 203), so as such, we deferred this specific change to later work. In our implementation Eval is a nullary constructor.

```
convEvalDataCon = pcSpecialDataCon convEvalDataConName []
                  callingConvTyCon
                  (CallingConv $ \_ -> [ConvEval])


convCallDataCon :: DataCon
convCallDataCon = pcSpecialDataCon convCallDataConName
                  [ mkListTy runtimeRepTy ]
                  callingConvTyCon (CallingConv prim_conv_fun)
 where
   prim_conv_fun [rr_ty_list]
     = ConvCall (concatMap (runtimeRepPrimRep doc) rr_tys)
     where
       rr_tys = extractPromotedList rr_ty_list
       doc    = text "convCallDataCon" <+> ppr rr_tys
   prim_conv_fun args
     = pprPanic "convCallDataCon" (ppr args)

convCallDataConTyCon :: TyCon
convCallDataConTyCon = promoteDataCon convCallDataCon

callingConvTyCon :: TyCon
callingConvTyCon = pcTyCon callingConvTyConName []
   [convEvalDataCon, convCallDataCon]

-- Haskell source equivalent
data CallingConv = ConvEval | ConvCall [RuntimeRep]
```

Figure 29: Definition of CallingConv and its constructors

Its constructors, convEvalDataCon and convCallDataCon are defined with the pcSpecialDataCon function which accepts a unique identifier, a list corresponding to the argument types of the constructor, the TyCon (or type) to which it belongs, and finally a term of type RuntimeRepInfo which encapsulates a function that converts the entire term to its corresponding primitive form for code generation purposes. In the case of the nullary ConvEval constructor, the list of argument types is an empty list. As ConvCall constructor accepts a list of RuntimeRep's its argument list is specified with (mkListTy runtimeRepTy). Both constructors belong to the CallinvConv type and therefore have callingConvTyCon as their third argument. Finally the CallingConv type is specified (as CallingConvTyCon) with the pcTyCon function which accepts a name, a list of type variables and a list of data constructors. We pass an empty list for the type variable argument as CallingConv is a completely monomorphic type, then we pass in our newly defined ConvEval and ConvCall constructors. The RuntimeInfo type is defined in a similar manner.

```
callingConvTy :: Type
callingConvTy = mkTyConTy callingConvTyCon

runtimeInfoTy :: Type
runtimeInfoTy = mkTyConTy runtimeInfoTyCon
```

These types are then promoted to the kind level using the mkTyConTy function.

4.4.2 PrimRep and RuntimeRepInfo

As mentioned in the previous section, our wired-in constructors encapsulate functionality for converting themselves to their primitive forms. These primitive forms are used in the code generation stage of the compiler and exist at a lower level of abstraction than RuntimeRep and CallingConv. For example, a RuntimeRep of the form (`TupleRep [IntRep, WordRep]`) would be converted to just (`[IntRep, WordRep]`) because an unboxed tuple is represented in memory as the just concatenation of its elements' representations. The primitive form of a representation is all the compiler needs to know in order to store the data in memory or a set of registers, for example (`[IntRep, WordRep]`) says that the value is stored in two different registers, one containing a machine integer and the other containing just one machine word.

Below in Fig. 30, is the definition of the RuntimeRepInfo type which encapsulates conversion functions from the level of RuntimeRep and CallingConv to their primitive equivalents (note that terms of RuntimeRep and CallingConv are of type Type as they are wired-in).

```
data RuntimeRepInfo
 = NoRRI
 | RuntimeRep ([Type] -> [PrimRep])
 | VecCount Int
 | VecElem PrimElemRep
 | RuntimeInfo ([Type] -> [PrimInfo])
 | CallingConv ([Type] -> [PrimConv])
```

Figure 30: The RuntimeRepInfo type

Though all wired-in constructors must be specified with an instance of RuntimeRepInfo, only a subset of special types which will be promoted to the kind level must contain this conversion functionality. All other non-promoted wired-in types can be specified using the NoRRI constructor of RuntimeRepInfo.

```
prim_conv_fun [rr_ty_list]
  = ConvCall (concatMap (runtimeRepPrimRep doc) rr_tys)
  where
    rr_tys = extractPromotedList rr_ty_list
    doc    = text "convCallDataCon" <+> ppr rr_tys
```

In the conversion function we specified for the ConvCall constructor (which accepts a list of RuntimeReps) we call runtimeRepPrimRep with each argument, concatenate the results and pass them to the primitive ConvCall constructor. The runtimeRepPrimRep function merely pattern matches on the RuntimeRepInfo argument of the data constructor and recursively calls its conversion function upon it.

4.4.3 GetRep and GetConv

In addition to wiring in our new types, we must also wire in the type families (GetRep and GetConv) we specified in the previous paragraphs such that they are available within core. They are defined in GHC.Builtin.Types in a manner similar to CallingConv and RuntimeInfo (shown below in Fig. 31).

```
getRepTyCon :: TyCon
getRepTyCon = mkFamilyTyCon getRepTyConName binders runtimeRepTy
                        Nothing
                        (BuiltInSynFamTyCon trivialBuiltInFamily)
                        Nothing
                        NotInjective
 where
   binders = mkTemplateAnonTyConBinders [runtimeInfoTy]


getConvTyCon :: TyCon
getConvTyCon = mkFamilyTyCon getConvTyConName binders callingConvTy
                        Nothing
                        (BuiltInSynFamTyCon trivialBuiltInFamily)
                        Nothing
                        NotInjective
 where
   binders = mkTemplateAnonTyConBinders [runtimeInfoTy]
```

Figure 31: The GetRep and GetConv type families expressed in Core

The mkFamilyTyCon (from GHC.Core.TyCon) accepts a name, a list of kind arguments, an output kind and some additional meta-data. Both GetRep and GetConv accept an argument of kind RuntimeInfo. And as their names would suggest, GetRep returns a type of kind RuntimeRep, and GetConv returns a type of kind RuntimeInfo.

One might ask why these type families need to exist within Core. Why couldn't we just destruct a RuntimeInfo and extract the appropriate field if we needed its value? The problem is that in many cases the field does not exist yet. If a type is still in the process of being inferred, it is possible that the RuntimeInfo we are attempting to extract a field from is merely a meta type variable (a placeholder type which will be inferred at a later time). Our representation accessor getRep', shown below in Fig. 32,

```
getRep' :: Type -> Type
getRep' rinfo
 | Just rep <- extractRep rinfo
 = rep
 | otherwise
 = mkTyConApp getRepTyCon [rinfo]
```

Figure 32: The GetRep simplification function

first attempts extract the RuntimeRep field of a RuntimeInfo (using the extractRep function) and if that fails (if for instance RuntimeInfo is a type variable),it returns the application of the getRepTyCon type constructor to the opaque rinfo, so that it can be beta-reduced at a later time when rinfo has been simplified (perhaps by substitution of a generic type variable or some other reductions inside rinfo) to a concretely constructed RuntimeInfo object.

```
getRep' (r1 :: RuntimeInfo)  --> GetRepTyCon [r1]
```

GHC preserves applications of type families as "coercions" that provide specific evidence that each change of type (be it by reducing a type family instance or by converting between two types that are equivalent at run-time but not compile-time), which is essential to the soundness of its type system[3]. For example, reducing the application of a type family function is not reduced by the type checker, but appears as an explicit coercion in the syntax of a program. Following suit, applications of GetRep and GetConv need to be preserved (and not beta-reduced) until code generation. This presented a problem in the isKindLevPoly function (the check for levity-polymorphism) which would return false positives in the presence of GetRep applications in the kind. To remedy this, we defined a simplifyRep function which would beta-reduce GetRep applications for the purposes of this check. It should be noted thatthis function MUST not be used in any context where a type is returned (i.e. as part of the typechecking pipeline), but merely within checks that return a boolean.

[3]This issue of type soundness was decided in discussions with Simon Peyton Jones and Richard Eisenberg.

4.5 Modifying TYPE and its Accessors

Next we modified the TYPE constructor to accept, instead of a type of kind RuntimeRep, a type of kind RuntimeInfo. The TYPE constructor is defined within GHC.Builtin.Types.Prim using the mkKindTyCon function found within GHC.Core.TyCon (shown below in Fig. 33).

```
tYPETyCon :: TyCon
tYPETyConName :: Name

tYPETyCon = mkKindTyCon tYPETyConName
                        (mkTemplateAnonTyConBinders [runtimeInfoTy])
                        liftedTypeKind
                        [Nominal]
                        (mkPrelTyConRepName tYPETyConName)
```

Figure 33: Definition of the TYPE constructor

mkKindTyCon accepts a unique identifier, a list of type arguments (which in this case is the singleton list containing our promoted RuntimeInfo type), a result kind, and some additional meta-data. Accordingly, we also have the new default (the definition of which is shown below in Fig. 34) kind which is defined using the buildSynTyCon function ("syn" for synonym) which accepts a name for the binding, a result kind (the default kind itself; hurray for laziness) and a right hand side for the binding; the right hand side of the synonym binding is a type constructor application of the TYPE constructor applied to LiftedRep and ConvEval, thus making our new default kind (`TYPE (RInfo LiftedRep ConvEval)`).

```
-- type Type = TYPE (‘RInfo ’LiftedRep ‘ConvEval)

liftedTypeKindTyCon :: TyCon
liftedTypeKindTyCon   = buildSynTyCon liftedTypeKindTyConName
                                      liftedTypeKind rhs
 where rhs = TyCoRep.TyConApp tYPETyCon [mkTyConApp
             runtimeInfoDataConTyCon
             [liftedRepTy, convEvalDataConTy]]
```

Figure 34: Definition of new default kind

Given the new kind structure, we must also change the signature of many existing type constructors such as the arrow (`->`) which constructs function types. The previous function type constructor accepts two RuntimeRep's (r1 and r2; the representations of the input and output types respectively) and constructs the resulting function type of the form (`TYPE r1 -> TYPE r2 -> Type`).

```
(->) :: forall (r1 :: RuntimeRep) (r2 :: RuntimeRep).
       TYPE r1 -> TYPE r2 -> Type
```

As such, the definition is RuntimeRep polymorphic. Given that our new TYPE constructor accepts a RuntimeInfo rather than a RuntimeRep, this definition of the arrow constructor will no longer be general enough. We could, for instance, use those RuntimeRep arguments to construct RuntimeInfo's to pass to the TYPE constructors in the resulting type with the ConvEval constructor hardcoded (as shown below).

```
(->) :: forall (r1 :: RuntimeRep) (r2 :: RuntimeRep).
      TYPE (RInfo r1 ConvEval) -> TYPE (RInfo r2 ConvEval) -> Type
```

This approach however, has two main drawbacks. Firstly, the definition of ConvEval will eventually be changed to contain a Levity field and once that change is made, this definition will no longer typecheck. Second, we may want to be polymorphic in representation as well as convention. Because we are merely constructing a type and don't need to generate code for invoking our convention polymorphic type, we are well within our rights to do; after all more polymorphism is always better.

```
(->) :: forall (r1 :: RuntimeRep) (c1 :: CallingConv)
               (r2 :: RuntimeRep) (c2 :: CallingConv).
      TYPE (RInfo r1 c1) -> TYPE (RInfo r2 c2) -> Type
```

We could then, for both input and output, quantify over both RuntimeRep and CallingConv to construct our RuntimeInfo types in the resulting function type. This approach though, as previously discussed, is overly verbose and more cumbersome to write out.

```
(->) :: forall (r1 :: RuntimeInfo) (r2 :: RuntimeInfo).
       TYPE r1 -> TYPE r2 -> Type
```

As RuntimeInfo is merely a product containing a RuntimeRep and a CallingConv, it would be semantically equivalent to merely quantify over a RuntimeInfo for each argument.

5 Results

Below in Fig. 35, we have the performance results of our worker-wrapper transformation according to NoFib; the official performance benchmarking test suite used by the GHC maintainers.

```
NoFib Results
--------------------------------------------------------------------------------
       Program           Size    Allocs   Runtime   Elapsed  TotalMem
--------------------------------------------------------------------------------
            CS          -----     -----     -----     -----     -----
           CSD          +0.0%      0.0%     +1.7%     +2.0%      0.0%
             S          +0.0%      0.0%     +1.0%     +1.1%      0.0%
          atom          +0.0%      0.0%     +1.9%     +1.9%      0.0%
        awards          +0.0%     +0.9%     0.000     0.000      0.0%
        banner          +0.0%     +0.0%     0.000     0.000      0.0%
  binary-trees          +0.0%     +0.0%     -0.4%     -0.4%      0.0%
        boyer2          +0.0%     +0.6%     0.005     0.005%     0.0%
      calendar          +0.0%     +0.0%     0.000     0.000%     0.0%
      cichelli          +0.0%      0.0%     0.028     0.028      0.0%
      clausify          +0.0%      0.0%     +3.9%     +3.9%      0.0%
     compress2          +0.0%     +3.8%     0.170     0.170      0.0%
  cryptarithm1           0.0%      0.0%     0.181     0.182      0.0%
  cryptarithm2          +0.0%      0.0%     0.003     0.003      0.0%
  digits-of-e1          +0.0%      0.0%     -3.2%     -3.1%      0.0%
  digits-of-e2          +0.0%     +0.0%     +0.2%     +0.2%      0.0%
         eliza          +0.0%     +0.5%     0.001     0.001      0.0%
        exp3_8          +0.0%      0.0%     +3.5%     +3.4%      0.0%
fannkuch-redux          +0.0%     +0.0%     +0.6%     +0.2%      0.0%
         fasta          +0.0%     +0.0%     +2.2%     +2.2%      0.0%
          fft2          +0.0%     +0.1%     +0.3%     +0.4%    +16.7%
        fulsom          +0.1%      0.0%     +2.6%     +2.6%      0.0%
        gamteb          +0.0%      0.0%     +1.5%     +1.4%     -3.4%
           gcd          +0.0%      0.0%     -4.7%     -4.6%      0.0%
   gen_regexps          +0.0%      0.0%     -0.1%     +0.0%      0.0%
       integer          +0.0%      0.0%     -4.7%     -4.5%      0.0%
  k-nucleotide          +0.2%  +2635.5%    +33.5%    +33.6%      0.0%
         kahan          +0.0%      0.0%     0.118     0.118      0.0%
       knights          +0.1%     +0.0%     +0.3%     +0.9%      0.0%
    last-piece          -0.0%     -2.0%     0.170     0.170      0.0%
          lcss          +0.0%      0.0%     +0.3%     +0.3%      0.0%
          life          +0.0%     +0.0%     0.082     0.082      0.0%
        linear          +0.1%     +0.0%     +3.2%     +3.4%    +33.3%
      maillist          +0.0%     +0.0%     0.017     0.017      0.0%
        mandel          +0.0%      0.0%     0.024     0.024      0.0%
       mandel2          +0.0%      0.0%     0.002     0.002      0.0%
       mkhprog          +0.0%     -0.0%     0.001     0.001      0.0%
        n-body          +0.0%      0.0%     +0.2%     +0.3%      0.0%
      nucleic2          +0.0%     +0.0%     0.027     0.027      0.0%
     paraffins          +0.0%     +0.0%     -1.8%     -1.7%     +2.3%
      pidigits          +0.0%      0.0%     +0.1%     -0.3%      0.0%
         power          +0.0%     -0.0%     -7.1%     -7.2%      0.0%
        pretty          +0.2%     +0.8%     0.000     0.000      0.0%
        primes          +0.0%      0.0%     -0.3%     -0.1%      0.0%
     primetest          +0.0%     +0.0%     -2.1%     -2.1%      0.0%
        puzzle          +0.1%      0.0%     0.047     0.048      0.0%
        queens          +0.0%      0.0%     -0.6%     -0.6%      0.0%
reverse-complem         +0.0%      0.0%     -4.2%     -3.9%      0.0%
          rfib          +0.0%      0.0%     -1.7%     -1.7%      0.0%
           rsa          +0.0%      0.0%     -3.6%     -3.8%      0.0%
         sched          +0.0%      0.0%     +2.0%     +2.1%      0.0%
       sorting          +0.0%      0.0%     0.001     0.001      0.0%
 spectral-norm          +0.0%      0.0%     +0.5%     +0.6%      0.0%
        sphere          +0.1%     -0.1%     +2.4%     +2.4%      0.0%
           tak          +0.0%      0.0%     +0.9%     +0.9%      0.0%
      treejoin          +0.0%      0.0%     0.041     0.041      0.0%
  wheel-sieve2          +0.0%      0.0%     +1.7%     +1.6%      0.0%
          x2n1          +0.0%      0.0%     0.168     0.168      0.0%
--------------------------------------------------------------------------------
           Min          -0.0%     -2.0%     -7.1%     -7.2%     -3.4%
           Max          +0.2%  +2635.5%    +33.5%    +33.6%    +33.3%
Geometric Mean          +0.0%     +6.1%     +0.7%     +0.7%     +0.8%
```

Figure 35: Performance benchmarking results

NoFib, for a series of Haskell programs, measures the size on disk, number of memory allocations, user running time, system running time and size in memory. The program runs the benchmark once without our optimization enabled and a second time with it enabled. It then compares the performance numbers of the two runs and outputs the change in percentage from the first run to the second. It is evident that there is an issue with our optimization. Though there are multiple cases such as reverse-complem, integer, and power which see speedups of 5-7%, there are also cases, most notably, k-nucleotide which see a dramatic increase in memory allocations (+2635.5%) and runtime (+33.5%). These pathological cases drive down our average speedup across all cases to -0.7%. After analysing the "optimized" Core outputs of these cases, we observed a combinatorial explosion of worker-wrapper transformation. That is to say that the interaction of our optimization with other worker-wrapper transformations (namely the strictness analysis pass) caused an exponential number of worker-wrappers to be allocated. As each new function created by the transformation corresponds to a thunk being allocated on the heap, the number of memory allocations for a few cases exploded by multiple orders of magnitude. Below, in Fig. 36, is an example of a worker-wrapper transformation that may be done by the strictness analysis optimization. The original "double" function accepts an Int which is represented by a pointer in the heap and doubles it.

```
double :: Int -> Int
double y = y * 2

-- optimization
double_wrapper :: Int -> Int
double_wrapper y =
  case y of
      I# x -> I# (double_worker x)

double_worker :: Int# -> Int#
double_worker x = timesInt# 2 x
```

Figure 36: Unboxing worker/wrapper transformation

The optimization then creates a worker-wrapper pair whose worker function operates over unboxed integers, meaning that instead of needing to follow a pointer into the heap, it deals with raw integer values which it extracts from a register. The wrapper uses pattern matching to obtain the raw integer value from the pointer and passes it to the worker. Just as we outlined in our worker-wrapper transformation, this wrapper can be inlined at all the call-sites of “double” then be beta-reduced to a call to the worker; thus transforming the original function to a more efficient version that deals with raw values. The problem occurs when our worker-wrapper transformation is run against this already optimized code. The effect, as shown below in Fig. 37, is that a new worker-wrapper pair is created for both the worker and the wrapper.

```
double_wr :: Int -> Int
double_wr = \y -> double_wk y

double_wk :: Int ~> Int
double = (\y ~>
  case y of
      I# x -> I# (double_worker x))

double_worker_wr :: Int# -> Int#
double_worker_wr x = \x -> double_worker_wk x

double_worker_wk :: Int# ~> Int#
double_worker = \x ~> timesInt# 2 x
```

Figure 37: Combinatorial explosion of worker/wrappers

For the cases where the wrappers can be successfully beta-reduced to the workers, though there would still be an increase in allocations as these extra definitions would still be floating around, the performance should not be affected negatively as the function calls would be short circuited to the innermost worker. If however, for some reason, the calls to the wrapper could not be beta reduced to a call to the worker, we would see an exponential increase in function calls.

6 Future Work

6.1 Worker/Wrapper

Our most pressing matter for future work is to eliminate (or mitigate) the worker/wrapper explosion issue discussed in the previous section. Our aim would be to combine the worker-wrapper transformations such that only a single worker-wrapper pair is created across all optimization passes.

In Fig. 38 below, the worker is both extensional and unboxed. Accordingly, the wrapper both unboxes the argument and uses the extensional function application to invoke the worker.

```
double :: Int -> Int
double y =
 case y of
     I# x -> I# (double_worker x)

double_worker :: Int# ~> Int#
double_worker = \x ~> timesInt# 2 x
```

Figure 38: Combining worker/wrapper transformations

This can either be accomplished by combining both optimizations into a single pass, or by allowing either pass to detect the presence of an existing worker/wrapper pair and modify them in place rather than creating a new pair. Our hope is that with this change we will eliminate the worker/wrapper explosion and therefore the dramatic increase in allocation seen in some of the test cases. If this issue is solved we are confident that the worker/wrapper transformation will see a net gain in performance.

Another bottleneck in performance is our. There exists within GHC, more robust arity analysis functionality which utilizes a graph based approach above the level of syntax trees. If we utilized this existing arity analysis infrastructure we would likely be able to achieve more aggressive lower bounds of arity to guide the worker/wrapper transformation.

6.2 Give Extensional Function to Users

Though the extensional arrow we have introduced is a feature of Haskell's intermediate language, there is nothing stopping us from making it a user facing feature; In fact there would be multiple advantages for doing so. Firstly, if a user were able to specify the arity of a function by manually specifying the function type signature with the extensional arrow, the user could rely on the guarantee that all calls to that function will be fully saturated at, thereby eliminating the need for dynamic arity checks at run-time or the worker/wrapper transformation at compile time . As there is an inherent cost associated with this transformation in increased allocations and possible increased function calls, bypassing it would allow for more dramatic speedups. On top of making some of the existing optimizations more efficient, it would enable optimizations which were previously not possible.

In the function appList shown at the top of in Fig. 39 below, it is currently unsound for the worker/wrapper transformation to convert the first argument to a list of arity two functions (as in the appList_w function at the bottom of Fig. 39). Given our current naive method of arity detection, we would look at the body of appList and deduce from the application of f that the first argument is a list of arity two functions. We would then use this arity to guide the creation of the new extensional worker function seen above. The problem is that there would be nothing stopping us from passing in a function such as g to our new extensional appList_w (in Fig. 39). The issue with this is that though the type of g may lead us to believe that it is an arity two function, in reality it is an arity one function which returns a closure. Although in the context of the original appList this difference would be immaterial as all normal functions behave that way, our new extensional appList expects x and y to be applied at the same time. This would cause a stack underflow at runtime if g were invoked as an arity two function. If however the user was able to specify appList at the source level as an extensional function, the typechecker would be able to verify that all functions in the last passed to appList would be of the form (a ¿ b ¿ c); thereby guaranteeing that all functions passed to it would be compiled to arity two functions.

```
appList :: [a -> a -> b] -> a -> [b]
appList [] x = []
appList (f:fs) x = f x x : appList fs x

appList_w :: [a ~> a \~> b] -> a -> [b]
appList_w [] x = []
appList_w (f:fs) x = f x x : appList fs x

g x = let z = expensive x in \y -> z + y

y = appList [g] 1
```

Figure 39: Unsound worker/wrapper transformation

Another bottleneck in performance is our ad-hoc, expression based arity analysis. There exists within GHC, more robust arity analysis functionality which utilizes a graph based approach above the level of syntax trees. If we utilized this existing arity analysis infrastructure we would likely be able to achieve more aggressive lower bounds of arity to guide the worker/wrapper transformation.

6.3 Finish changes to Kinds

As we noted in a previous section, we deferred the extraction of levity from the RuntimeRep kind to CallingConv. Once that area of the codebase has settled down, this change can be made more conveniently. This roadblock however, illuminates a problem with the kind interface, in that the user interface for specifying kinds is very tightly coupled to the inner-structure of kinds (ie the TYPE constructor, RuntimeRep and CallingConv). If for example we wanted to specify a function which ranged over all lifted types, our type signature would look something like this:

```
f :: (c :: CallingConv) (a :: TYPE (RInfo LiftedRep c))
f x = ...
```

The specification of this type depends heavily on the structure TYPE, RInfo, RuntimeRep and CallingConv (not to mention it's overly verbose). If for example we were to implement the change to RuntimeRep specified in Kinds are Calling Conventions[1] and extract levity to the calling convention field, the same kind specification would look something like this:

```
f :: (r :: RuntimeRep) (a :: TYPE (RInfo r (ConvEval Lifted)))
f x = ..
```

Instead of quantifying over CallingConv, we now must quantify over RuntimeRep. This would therefore be a breaking change for many users. It would be better to instead provide an interface to users which is more loosely coupled from the internal structure of kinds.

```
type IsPtr (a :: TYPE r) = Rep r ~ BoxedRep

f :: IsPtr a => a -> b
f x = ...
```

We could, for example use type constraints, as above, to specify that a kind is lifted without needing to construct a RuntimeInfo type in its signature. The IsLifted type constructor above accepts a type and utilizes the GetLevity type family to verify that the type's kind is lifted. With this approach, if the levity information were moved within the structure of kinds, we would merely need to update our definition of GetLevity and leave the user none the wiser.

REFERENCES CITED

[1] Paul Downen et al. *Making a faster curry with extensional types.* Aug. 2019. URL: `https://www.microsoft.com/en-us/research/publication/making-a-faster-curry-with-extensional-types/`.

[2] Paul Downen et al. “Kinds are calling conventions”. In: *International Conference on Functional Programming (ICFP’20).* ACM. ACM, Aug. 2020. URL: `https://www.microsoft.com/en-us/research/publication/kinds-are-calling-conventions/`.