title	author	description	first-written	last-updated	last-reviewed
Primitive Haskell	Michael Snoyman	Overview of peeling back layers of abstraction in GHC Haskell	2015-02-24	2015-02-24	2015-02-24

The point of this chapter is to help you peel back some of the layers of abstraction in Haskell coding, with the goal of understanding things like primitive operations, evaluation order, and mutation. Some concepts covered here are generally "common knowledge" in the community, while others are less well understood. The goal is to cover the entire topic in a cohesive manner. If a specific section seems like it's not covering anything you don't already know, skim through it and move on to the next one.

While this chapter is called "Primitive Haskell," the topics are very much GHC-specific. I avoided calling it "Primitive GHC" for fear of people assuming it was about the internals of GHC itself. To be clear: these topics apply to anyone compiling their Haskell code using the GHC compiler.

Note that we will not be fully covering all topics here. There is a "further reading" section at the end of this chapter with links for more details.

Let's do addition

Let's start with a really simple question: tell me how GHC deals with the expression `1 + 2`. What *actually* happens inside GHC? Well, that's a bit of a trick question, since the expression is polymorphic. Let's instead use the more concrete expression `1 + 2 :: Int`.

The `+` operator is actually a method of the `Num` type class, so we need to look at the `Num Int` instance:

```
instance  Num Int  where
    I# x + I# y = I# (x +# y)
```

Huh... well *that* looks somewhat magical. Now we need to understand both the `I#` constructor and the `+#` operator (and what's with the hashes all of a sudden?). If we do a Hoogle search, we can easily find the relevant docs, which leads us to the following definition:

```
data Int = I# Int#
```

So our first lesson: the `Int` data type you've been using since you first started with Haskell isn't magical at all, it's defined just like any other algebraic data type... except for those hashes. We can also search for `+#`, and end up at some documentation giving the type:

```
+# :: Int# -> Int# -> Int#
```

Now that we know all the types involved, go back and look at the `Num` instance I quoted above, and make sure you feel comfortable that all the types add up (no pun intended). Hurrah, we now understand exactly how addition of `Int`s works. Right?

Well, not so fast. The Haddocks for `+#` have a very convenient source link... which (apparently due to a Haddock bug) doesn't actually work. However, it's easy enough to find the correct hyperlinked source. And now we see the implementation of `+#`, which is:

```
infixl 6 +#
(+#) :: Int# -> Int# -> Int#
(+#) = let x = x in x
```

That doesn't look like addition, does it? In fact, `let x = x in x` is another way of saying bottom, or `undefined`, or infinite loop. We have now officially entered the world of primops.

primops

primops, short for primary operations, are core pieces of functionality provided by GHC itself. They are the magical boundary between "things we do in Haskell itself" and "things which our implementation provides." This division is actually quite elegant; as we already explored, the standard `+` operator and `Int` data type you're used to are actually themselves defined in normal Haskell code, which provides many benefits: you get standard type class support, laziness, etc. We'll explore some of that in more detail later.

Look at the implementation of other functions in `GHC.Prim`; they're *all* defined as `let x = x in x`. When GHC reaches a call to one of these primops, it automatically replaces it with the real implementation for you, which will be some assembly code, LLVM code, or something similar.

You may be wondering: why bother with this dummy implementation at all? The sole reason is to give Haddock documentation for the primops a place to live. GHC.Prim is processed by Haddock more or less like any other module; but is effectively ignored by GHC itself.

Why do all of these functions end in a `#`? That's called the magic hash (enabled by the `MagicHash` language extension), and it is a convention to distinguish boxed and unboxed types and operations. Which, of course, brings us to our next topic.

Unboxed types

The `I#` constructor is actually just a normal data constructor in Haskell, which happens to end with a magic hash. However, `Int#` is *not* a normal Haskell data type. In `GHC.Prim`, we can see that it's implementation is:

```
data Int#
```

Which, like everything else in `GHC.Prim` is really a lie. In fact, it's provided by the implementation, and is in fact a normal `long int` from C (32-bit or 64-bit, depending on architecture). We can see something even funnier about it in GHCi:

```
> :k Int
Int :: *
> :k Int#
Int# :: #
```

That's right, `Int#` has a different *kind* than normal Haskell datatypes: `#`. To quote the GHC docs:

> Most types in GHC are boxed, which means that values of that type are represented by a pointer to a heap object. The representation of a Haskell `Int`, for example, is a two-word heap object. An unboxed type, however, is represented by the value itself, no pointers or heap allocation are involved.

See those docs for more information on distinctions between boxed and unboxed types. It is vital to understand those differences when working with unboxed values. However, we're not going to go into those details now. Instead, let's sum up what we've learnt so far:

- `Int` addition is just normal Haskell code in a typeclass
- `Int` itself is a normal Haskell datatype
- GHC provides `Int#` and `+#` as an unboxed `long int` and addition on that type, respectively. This is exported by `GHC.Prim`, but the real implementation is "inside" GHC.
- An `Int` contains an `Int#`, which is an unboxed type.
- Addition of `Int`s takes advantage of the `+#` primop.

More addition

Alright, we understand basic addition! Let's make things a bit more complicated. Consider the program:

```
main = do
    let x = 1 + 2
        y = 3 + 4
    print x
    print y
```

We know for certain that the program will first print `3`, and then print `7`. But let me ask you a different question. Which operation will GHC perform first: `1 + 2` or `3 + 4`? If you guessed `1 + 2`, you're *probably* right, but not necessarily! Thanks to referential transparency, GHC is fully within its rights to rearrange evaluation of those expressions and add `3 + 4` before `1 + 2`. Since neither expression depends on the result of the other, we know that it is irrelevant which evaluation occurs first.

Note: This is covered in much more detail on the GHC wiki's evaluation order and state tokens page.

That begs the question: if GHC is free to rearrange evaluation like that, how could I say in the previous paragraph that the program will always print `3` before printing `7`? After all, it doesn't appear that `print y` uses the result of `print x` at all, so we not rearrange the calls? To answer that, we again need to unwrap some layers of abstraction. First, let's evaluate and inline `x` and `y` and get rid of the `do`-notation sugar. We end up with the program:

```
main = print 3 >> print 7
```

We know that `print 3` and `print 7` each have type `IO ()`, so the `>>` operator being used comes from the `Monad IO` instance. Before we can understand that, though, we need to look at the definition of `IO` itself

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

We have a few things to understand about this line. Firstly, `State#` and `RealWorld`. For now, just pretend like they are a single type; we'll see when we get to `ST` why `State#` has a type parameter.

The other thing to understand is that `(# ... #)` syntax. That's an *unboxed tuple*, and it's a way of returning multiple values from a function. Unlike a normal, boxed tuple, unboxed tuples involve no extra allocation and create no thunks.

So `IO` takes a real world state, and gives you back a real world state and some value. And that right there is how we model side effects and mutation in a referentially transparent language. You may have heard the description of `IO` as "taking the world and giving you a new one back." What we're doing here is threading a specific state token through a series of function

calls. By creating a dependency on the result of a previous function, we are able to ensure evaluation order, yet still remain purely functional.

Let's see this in action, by coming back to our example from above. We're now ready to look at the `Monad IO` instance:

```
instance  Monad IO  where
    (>>) = thenIO

thenIO :: IO a -> IO b -> IO b
thenIO (IO m) k = IO $ \ s -> case m s of (# new_s, _ #) -> unIO k new_s

unIO :: IO a -> (State# RealWorld -> (# State# RealWorld, a #))
unIO (IO a) = a
```

(Yes, I changed things a bit to make them easier to understand. As an exercise, compare that this version is in fact equivalent to what is actually defined in `GHC.Base`.)

Let's inline these definitions into `print 3 >> print 7`:

```
main = IO $ \s0 ->
    case unIO (print 3) s0 of
        (# s1, res1 #) -> unIO (print 7) s1
```

Notice how, even though we ignore the *result* of `print 3` (the `res1` value), we still depend on the new state token `s1` when we evaluate `print 7`, which forces the order of evaluation to first evaluate `print 3` and then evaluate `print 7`.

If you look through `GHC.Prim`, you'll see that a number of primitive operations are defined in terms of `State# RealWorld` or `State# s`, which allows us to force evaluation order.

Exercise: implement a function `getMaskingState :: IO Int` using the `getMaskingState#` primop and the `IO` data constructor.

The ST monad

Let's compare the definitions of the `IO` and `ST` types:

```
newtype IO   a = IO (State# RealWorld -> (# State# RealWorld, a #))
newtype ST s a = ST (State# s         -> (# State# s,         a #))
```

Well *that* looks oddly similar. Said more precisely, `IO` is isomorphic to `ST RealWorld`. `ST` works under the exact same principles as `IO` for threading state through, which is why we're able to have things like mutable references in the `ST` monad.

By using an uninstantiated `s` value, we can ensure that we aren't "cheating" and running arbitrary `IO` actions inside an `ST` action. Instead, we just have "local state" modifications, for some definition of local state. The details of using `ST` correctly and the Rank2Types approach to `runST` are interesting, but beyond the scope of this chapter, so we'll stop discussing them here.

Since `ST RealWorld` is isomorphic to `IO`, we should be able to convert between the two of them. `base` does in fact provide the `stToIO` function.

Exercise: write a pair of functions to convert between `IO a` and `ST RealWorld a`.

Exercise: `GHC.Prim` has a section on mutable variables, which forms the basis on `IORef` and `STRef`. Provide a new implementation of `STRef`, including `newSTRef`, readSTRef, and writeSTRef`.

PrimMonad

It's a bit unfortunate that we have to have two completely separate sets of APIs: one for `IO` and another for `ST`. One common example of this is `IORef` and `STRef`, but- as we'll see at the end of this section- there are plenty of operations that we'd like to be able to generalize.

This is where `PrimMonad`, from the `primitive` package, comes into play. Let's look at its definition:

```
-- | Class of primitive state-transformer monads
class Monad m => PrimMonad m where
  -- | State token type
  type PrimState m

  -- | Execute a primitive operation
  primitive :: (State# (PrimState m) -> (# State# (PrimState m), a #)) -> m a
```

Note: I have *not* included the `internal` method, since it will likely be removed. In fact, at the time you're reading this, it may already be gone!

`PrimState` is an associated type giving the type of the state token. For `IO`, that's `RealWorld`, and for `ST s`, it's `s`. `primitive` gives a way to lift the internal implementation of both `IO` and `ST` to the monad under question.

Exercise: Write implementations of the `PrimMonad IO` and `PrimMonad (ST s)` instances, and compare against the real ones.

The primitive package provides a number of wrappers around types and functions from `GHC.Prim` and generalizes them to both `IO` and `ST` via the `PrimMonad` type class.

Exercise: Extend your previous `STRef` implementation to work in any `PrimMonad`. After you're done, you may want to have a look at Data.Primitive.MutVar.

The `vector` package builds on top of the `primitive` package to provide mutable vectors that can be used from both `IO` and `ST`. This chapter is *not* a tutorial on the `vector` package, so we won't go into any more details now. However, if you're curious, please look through the `Data.Vector.Generic.Mutable` docs.

ReaderIO monad

To tie this off, we're going to implement a `ReaderIO` type. This will flatten together the implementations of `ReaderT` and `IO`. Generally speaking, there's no advantage to doing this: GHC should always be smart enough to generate the same code for this and for `ReaderT r IO` (and in my benchmarks, they perform identically). But it's a good way to test that you understand the details here.

You may want to try implementing this yourself before looking at the implementation below.

```
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE MagicHash             #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE UnboxedTuples         #-}
import Control.Applicative        (Applicative (..))
import Control.Monad              (ap, liftM)
import Control.Monad.IO.Class     (MonadIO (..))
import Control.Monad.Primitive    (PrimMonad (..))
import Control.Monad.Reader.Class (MonadReader (..))
import GHC.Base                   (IO (..))
import GHC.Prim                   (RealWorld, State#)
```

```haskell
-- | Behaves like a @ReaderT r IO a@.
newtype ReaderIO r a = ReaderIO
    (r -> State# RealWorld -> (# State# RealWorld, a #))

-- standard implementations...
instance Functor (ReaderIO r) where
    fmap = liftM
instance Applicative (ReaderIO r) where
    pure = return
    (<*>) = ap

instance Monad (ReaderIO r) where
    return x = ReaderIO $ \_ s -> (# s, x #)
    ReaderIO f >>= g = ReaderIO $ \r s0 ->
        case f r s0 of
            (# s1, x #) ->
                let ReaderIO g' = g x
                 in g' r s1

instance MonadReader r (ReaderIO r) where
    ask = ReaderIO $ \r s -> (# s, r #)
    local f (ReaderIO m) = ReaderIO $ \r s -> m (f r) s

instance MonadIO (ReaderIO r) where
    liftIO (IO f) = ReaderIO $ \_ s -> f s

instance PrimMonad (ReaderIO r) where
    type PrimState (ReaderIO r) = RealWorld

    primitive f = ReaderIO $ \_ s -> f s

    -- Cannot properly define internal, since there's no way to express a
    -- computation that requires an @r@ input value as one that doesn't. This
    -- limitation of @PrimMonad@ is being addressed:
    --
    -- https://github.com/haskell/primitive/pull/19
    internal (ReaderIO f) =
        f (error "PrimMonad.internal: environment evaluated")
```

Exercise: Modify the `ReaderIO` monad to instead be a `ReaderST` monad, and take an `s` parameter for the specific state token.

Further reading

- GHC docs on primitives
- GHC Wiki on PrimOps
- Evaluation order and state tokens