rio: A standard library

Haskell is a powerful and flexible language. It allows you to solve common programming problems with a plethora of approaches. It is possible to almost unceasingly innovate in both library design and overall code design.

On the other end of this spectrum, we have the libraries that ship with GHC, especially `base`. There is a strong sentiment for maintaining backwards compatibility and retaining workflows developed over the decades of the Haskell language. This includes some great stuff, but also some things many would consider design mistakes (such as partial functions).

We're left with a world where there are arguably too many degrees of freedom when starting a new Haskell project, and too many points of failures by relying on dangerous code. Teams can easily spend significant time making basic architectural and library decisions on a new project. And for those relatively new to the Haskell ecosystem, it's all too easy to make choices with unknown costs, with those costs only showing themselves later.

The overall goal of the documentation here is to help avoid these kinds of situations, by providing opinionated, well tested advice. The `rio` library is our best shot at codifying large parts of that advice.

We'll document the details through the rest of this document. But in short: if you use the `RIO` module as a replacement for `Prelude` as demonstrated, you will automatically bypass many pitfalls in Haskell coding. By giving up some of the degrees of freedom granted by Haskell, you'll be able to focus instead on solving your actual coding challenge.

`rio` quick start

As usual, you'll need to depend on the `rio` library to use it. In a typical project, this will mean adding `rio` to your `package.yaml` or `.cabal` file's dependencies list. If using something like the [Stack script interpreter](), this will happen automatically. Next, you'll want to use the `RIO` module as your replacement prelude by adding the language extension:

```
{-# LANGUAGE NoImplicitPrelude #-}
```

You'll likely want to include some other commonly used extensions as well, like `OverloadedStrings`. You can find a [list of recommended extensions]().

Next, add an import of the `RIO` module:

```
import RIO
```

When you need functionality for other data types like `ByteString` and `Text`, we recommend importing qualified from the `RIO` version of the module using the recommended qualified name. This avoids the need for modifying your `package.yaml` file for each new usage, ensures that only safe functions are imported by default, and ensures consistency across `rio`-using projects. For example:

```
import qualified RIO.ByteString as B
```

Not much to it, but to ensure your libraries are set up correctly, run this:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = runSimpleApp $ logInfo "Hello World!"
```

A standard library

`rio`'s tag line is "a standard library for Haskell." The idea is that the *actual* standard library for Haskell—the `base` package—leaves much to be desired. Very few real world applications use only `base`. There is a common set of functionality that is used by a large majority of applications. Standard libraries in other languages (such as Rust) are more "batteried included." `rio` attempts to:

- Pull in the commonly used libraries in the rest of the ecosystem
- Remove some functionality in `base` and other libraries which we deem as a non-best practice (like partial functions)
- Overall avoid innovation (though we do have a bit of innovation in the library where necessary)

That last point is part of what distinguishes `rio` from similar efforts. Also, `rio` to some extent supersedes previous efforts like `classy-prelude`, which attempted to use typeclass generalization instead of qualified imports. The Haskell community has overall decided on qualified imports and monomorphic functions; `rio` embraces this approach.

The `RIO` type

One of the hallmarks of the `rio` package is, unsurprisingly, the `RIO` data type. You can fully use the library without embracing the `RIO` data type (see the "lifting and unlifting" section below). But we recommend trying out the `RIO` type as well.

The `RIO` data type is based entirely on the `ReaderT` design pattern. I won't rehash that blog post here. I will say that in many commercial projects the FP Complete team has worked on, using `RIO` has short-circuited long design discussions around monad transformer stacks. And conversely, projects we've assisted on which have used their own monad transformer or effects approaches have often spent significant time on designing and debugging.

That said: this article isn't designed to convince you of anything, the aforementioned blog post is intended to do that! This article is intended to show you how to get stuff done. So if you're unconvinced, please suspend disbelief for now and continue with the rest of this tutorial.

The `RIO` data type looks like this:

```
newtype RIO env a = RIO (ReaderT env IO a)
```

Each time you see `RIO env a`, you can mentally convert it to `env -> IO a`, or "this thing has some input `env` and can perform arbitrary `IO` actions." This is deceptively simple, because this one approach allows us for a lot of flexibility.

Simple environments

Consider this trivial program that doesn't use `rio` at all:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
main :: IO ()
main = do
  let name = "Alice"
  sayHello name
  sayGoodbye name

sayHello :: String -> IO ()
sayHello name = putStrLn $ "Hello, " ++ name

sayGoodbye :: String -> IO ()
sayGoodbye name = putStrLn $ "Goodbye, " ++ name
```

Fairly straightforward. But perhaps we find it tedious to manually pass around the name. In an example this short, that's a silly complaint, but as we'll see below in large applications with lots of context, it's a real concern. The `RIO` type allows us to handle this:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import Prelude (putStrLn) -- we'll explain why we need this in logging
import RIO

type Name = String

main :: IO ()
main = do
  let name = "Alice"
  runRIO name $ do
    sayHello
    sayGoodbye

sayHello :: RIO Name ()
sayHello = do
  name <- ask
  liftIO $ putStrLn $ "Hello, " ++ name

sayGoodbye :: RIO Name ()
sayGoodbye = do
  name <- ask
  liftIO $ putStrLn $ "Goodbye, " ++ name
```

For an example this size, the change isn't at all warranted. But it *does* demonstrate the basic pattern with `RIO`:

- Put commonly used data in an environment
- Use `ask` to get the environment
- use `liftIO` to run underlying `IO` actions

Now let's make our example a little bit more compelling. We currently print to `stdout` by using `putStrLn`. Instead, I'd like to make the choice of output handle configurable. Without `RIO`, we could do this by having `sayHello` and `sayGoodbye` take a `Handle` as a parameter, e.g.:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import System.IO (hPutStrLn, stderr)

type Name = String

main :: IO ()
main = do
  let name = "Alice"
  runRIO name $ do
    sayHello stderr
    sayGoodbye stderr

sayHello :: Handle -> RIO Name ()
sayHello h = do
  name <- ask
  liftIO $ hPutStrLn h $ "Hello, " ++ name

sayGoodbye :: Handle -> RIO Name ()
sayGoodbye h = do
  name <- ask
  liftIO $ hPutStrLn h $ "Goodbye, " ++ name
```

But instead, we're going to be a bit fancier, and declare a proper application environment type:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import System.IO (hPutStrLn, stderr)

data App = App
  { appName :: !String
  , appHandle :: !Handle
  }

main :: IO ()
main = do
  let app = App
        { appName = "Alice"
        , appHandle = stderr
        }
  runRIO app $ do
    sayHello
    sayGoodbye

sayHello :: RIO App ()
sayHello = do
  App name h <- ask
  liftIO $ hPutStrLn h $ "Hello, " ++ name

sayGoodbye :: RIO App ()
sayGoodbye = do
  App name h <- ask
  liftIO $ hPutStrLn h $ "Goodbye, " ++ name
```

At this point, `RIO` is starting to look a little bit more compelling.

Exercise Define a helper function of type `String -> RIO App ()` and use it in `sayHello` and `sayGoodbye` instead of calling `hPutStrLn` directly.

Solution

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import System.IO (hPutStrLn, stderr)

data App = App
  { appName :: !String
  , appHandle :: !Handle
  }

main :: IO ()
main = do
  let app = App
        { appName = "Alice"
        , appHandle = stderr
        }
  runRIO app $ do
    sayHello
    sayGoodbye

say :: String -> RIO App ()
say msg = do
  App _name h <- ask
  liftIO $ hPutStrLn h msg

sayHello :: RIO App ()
sayHello = do
  App name _h <- ask
  say $ "Hello, " ++ name

sayGoodbye :: RIO App ()
sayGoodbye = do
  App name _h <- ask
  say $ "Goodbye, " ++ name
```

Has type classes

Now we'd like to tell the user what time it is. We'll modify our program a bit further:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import RIO.Time (getCurrentTime)
import System.IO (hPutStrLn, stderr)

data App = App
  { appName :: !String
  , appHandle :: !Handle
  }

main :: IO ()
main = do
  let app = App
        { appName = "Alice"
        , appHandle = stderr
        }
  runRIO app $ do
    sayHello
    sayTime
    sayGoodbye

say :: String -> RIO App ()
say msg = do
  App _name h <- ask
  liftIO $ hPutStrLn h msg

sayHello :: RIO App ()
sayHello = do
  App name _h <- ask
  say $ "Hello, " ++ name

sayTime :: RIO App ()
sayTime = do
  now <- getCurrentTime
  say $ "The time is: " ++ show now

sayGoodbye :: RIO App ()
sayGoodbye = do
  App name _h <- ask
  say $ "Goodbye, " ++ name
```

There's a bit of a problem here though. The `sayTime` action requires an `App` environment. However, it never actually uses the `appName` field. This can make it harder to use in a situation where there is no actual name. Also, if we wanted to define this in a library to be used in multiple projects, we'd have to hard-code this one specific application type, which wouldn't apply to many applications. (This also applies to our `say` function.) We need something more flexible.

The approach we recommend in `rio` is to define helper `Has*` typeclasses, and define library functions in terms of those. I could describe the technique, but it's really much simpler to just demonstrate it:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import RIO.Time (getCurrentTime)
import System.IO (hPutStrLn, stderr, stdout)

data App = App
  { appName :: !String
  , appHandle :: !Handle
  }

class HasHandle env where
  getHandle :: env -> Handle
instance HasHandle Handle where
  getHandle = id
instance HasHandle App where
  getHandle = appHandle

main :: IO ()
main = do
  let app = App
        { appName = "Alice"
        , appHandle = stderr
        }
  runRIO app $ do
    sayHello
    sayTime
    sayGoodbye
  -- Also works!
  runRIO stdout sayTime

say :: HasHandle env => String -> RIO env ()
say msg = do
  env <- ask
  liftIO $ hPutStrLn (getHandle env) msg

sayTime :: HasHandle env => RIO env ()
sayTime = do
  now <- getCurrentTime
  say $ "The time is: " ++ show now

sayHello :: RIO App ()
sayHello = do
  App name _h <- ask
  say $ "Hello, " ++ name

sayGoodbye :: RIO App ()
sayGoodbye = do
  App name _h <- ask
  say $ "Goodbye, " ++ name
```

We define the `HasHandle` typeclass, and then replace `App` in `say` and `sayTime` with a type variable `env`. Then we use the `getHandle` method instead of the `appHandle` accessor. Voila!

Exercise Define a new data type, `App2`, such that the following code works:

```
let app2 = App2 { app2Handle = stdout, app2FavoriteColor = "red" }
runRIO app2 sayTime
```

Bonus points for writing a `sayFavoriteColor` action too.

Boilerplate

It's worth taking a break here to address an elephant in the room. Defining these typeclasses and providing instances is boilerplate. It's slightly tedious. The gut reaction of many Haskellers may be to find some way to automate around this, with some `Generic`s usage, or Template Haskell, or something else. I recommend against that for multiple reasons:

- The boilerplate here, amortized across a project, is really small
- This is the "safe" kind of boilerplate: the compiler will almost always prevent you from making a mistake
- The code above is obvious and easy to follow
- The code above compiles really quickly

Again, this tutorial isn't about trying to convince anyone of anything. This is the recommended practice. I'm only including this section since it's such a common objection from Haskellers.

Also, the next section makes the boilerplate slightly more annoying. So be it :).

Lenses

Let's say halfway through your application, you'd like to switch to a different output handle. You can do that to some extent with the above.

Exercise Make this code compile:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import RIO.Time (getCurrentTime)
import System.IO (hPutStrLn, stderr, stdout)

data App = App
  { appName :: !String
  , appHandle :: !Handle
  }

class HasHandle env where
  getHandle :: env -> Handle
instance HasHandle Handle where
  getHandle = id
instance HasHandle App where
  getHandle = appHandle

main :: IO ()
main = do
  let app = App
        { appName = "Alice"
        , appHandle = stderr
        }
  runRIO app $ do
    switchHandle stdout sayHello
    sayTime

switchHandle :: Handle -> RIO App a -> RIO App a
switchHandle h inner = _

say :: HasHandle env => String -> RIO env ()
say msg = do
  env <- ask
  liftIO $ hPutStrLn (getHandle env) msg

sayTime :: HasHandle env => RIO env ()
sayTime = do
  now <- getCurrentTime
  say $ "The time is: " ++ show now

sayHello :: RIO App ()
sayHello = do
  App name _h <- ask
  say $ "Hello, " ++ name
```

Solution We can do this with a combination of `ask` and `runRIO` inside `switchHandle`:

```haskell
switchHandle :: Handle -> RIO App a -> RIO App a
switchHandle h inner = do
  app <- ask
  let app' = app { appHandle = h }
  runRIO app' inner
```

Unfortunately, we needed to hardcode `switchHandle` to work on the `App` data type instead of using the `HasHandle` typeclass. The reason for this is simple: the `HasHandle` typeclass provides the method `getHandle`, which allows us to *view* the `Handle`, but not *set* it. We could provide an additional `setHandle` method in the typeclass. If you'd like to do that as an exercise, feel free. However, we're going to jump straight ahead to the recommended solution: lenses.

The basic concept of lenses is to package together a getter and setter for a field in a data structure. There is *lots* to discuss around lenses: deeply nested fields, composition, prisms/folds/traversals, laws, and more. This section is not a general tutorial on lenses, not by a long shot. Instead, we're going to demonstrate in a cookbook style how to use the subset of lenses necessary to effectively use `RIO`. Feel free to [read more about lens](#) in its tutorial.

What we need is a lens that lets us peek from our big environment type into a `Handle`. This is `Lens' env Handle`. (The tick at the end of `Lens'` means "simple lens," in that it doesn't change any type parameters. Again, see the tutorial above for more information.) We can replace `getHandle` in our typeclass with:

```
class HasHandle env where
  handleL :: Lens' env Handle
```

Defining the instance for `Handle` is pretty cute:

```
instance HasHandle Handle where
  handleL = id
```

The `App` instance is a bit more involved. Since this is cookbook-style, I'll give you the code without deep explanation. You'll end up seeing this pattern often in `RIO` code.

```
instance HasHandle App where
  handleL = lens appHandle (\x y -> x { appHandle = y })
```

Now we can modify our `switchHandle` function to work on any `HasHandle` instance:

```
switchHandle :: HasHandle env => Handle -> RIO env a -> RIO env a
switchHandle h inner = do
  env <- ask
  let env' = set handleL h env
  runRIO env' inner
```

And similarly we can modify the `say` function to use `handleL` instead of `getHandle`:

```
say :: HasHandle env => String -> RIO env ()
say msg = do
  env <- ask
  let h = view handleL env
  liftIO $ hPutStrLn h msg
```

Cool, but let's get a bit more clever on both of these. We can leverage the fact that `RIO` is a `MonadReader` instance and use the `local` function to modify the environment, and the `view` function to bypass the usage of `ask`:

```haskell
switchHandle :: HasHandle env => Handle -> RIO env a -> RIO env a
switchHandle h = local (set handleL h)

say :: HasHandle env => String -> RIO env ()
say msg = do
  h <- view handleL
  liftIO $ hPutStrLn h msg
```

Now we're cooking with gas!

Exercise Modify the program below, without changing the signature of `addLastName`, so that the output of the program is "Hello, Alice Smith".

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
import RIO
import System.IO (hPutStrLn, stderr, stdout)

data App = App
  { appName :: !String
  , appHandle :: !Handle
  }

class HasHandle env where
  handleL :: Lens' env Handle
instance HasHandle App where
  handleL = lens appHandle (\x y -> x { appHandle = y })

class HasName env where
  nameL :: env -- change this!
instance HasName App where
  nameL = _

main :: IO ()
main = do
  let app = App
        { appName = "Alice"
        , appHandle = stderr
        }
  runRIO app $ addLastName sayHello

addLastName :: HasName env => RIO env a -> RIO env a
addLastName = _

say :: HasHandle env => String -> RIO env ()
say msg = do
  h <- view handleL
  liftIO $ hPutStrLn h msg

sayHello :: RIO App ()
sayHello = do
  App name _h <- ask
  say $ "Hello, " ++ name
```

Solution We need a proper type for `nameL`:

```
class HasName env where
  nameL :: Lens' env String
```

The typeclass instance for `App` is then fairly boilerplate:

```
instance HasName App where
  nameL = lens appName (\x y -> x { appName = y })
```

Implementing `addLastName` can be done a few ways. Using `view` and `set`, it can look like this:

```
addLastName :: HasName env => RIO env a -> RIO env a
addLastName inner = do
  name <- view nameL
  let name' = name ++ " Smith"
  env <- ask
  let env' = set nameL name' env
  runRIO env' inner
```

Instead of the `ask` and `runRIO` calls, we can instead use `local`:

```
addLastName :: HasName env => RIO env a -> RIO env a
addLastName inner = do
  name <- view nameL
  let name' = name ++ " Smith"
  local (set nameL name') inner
```

Or, we can introduce a new lens function, `over`, which combines both `view` and `set`.

```
addLastName :: HasName env => RIO env a -> RIO env a
addLastName = local (over nameL (++ " Smith"))
```

Personally, I find the name "modify" a bit more intuitive, but `modify` was already taken by the `mtl` library when lens was written.

Logging

Way, way, way above you may have noticed that we had to import `putStrLn` from `Prelude`. Why is such a basic, vital function not exposed from the `RIO` module itself? The first reason is that it is *inefficient*, being built on top of `String`s. But if that was the only concern, we could simply export a more efficient version built on top of a better data type.

This is part of a deeper design decision of `rio`. In general, we claim that there are basically two categories of console output:

- User targeted, textual output. We generally call this *logging*. This needs to take into account things like the console's character encoding.
- Machine targeted output, either textual or binary. In either case, this kind of output should *not* take into account the console's character encoding, but instead provide consistent output across systems.

This concept likely deserves a deeper explanation in the `rio` context. The best I can give for now is a link to the beware of `readFile` blog post, which captures much of the idea.

The `putStrLn` function blurs the line between these two concepts. This isn't specific to `putStrLn`; most standard libraries in most languages somewhat blur this line. In `rio`, we try to be more explicit. And so we have a set of logging functions.

Let's first see a working example, and then break it!

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = runSimpleApp $ logInfo "Hello World!"
```

The `runSimpleApp` function sets up an environment (called `SimpleApp`) and calls `runRIO` on it. Let's see what happens if we have a dummy environment instead:

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = runRIO () $ logInfo "Hello World!"
```

On my machine, we get the error message:

```
Main.hs:8:20: error:
    • No instance for (HasLogFunc ()) arising from a use of ‘logInfo’
    • In the second argument of ‘($)’, namely ‘logInfo "Hello World!"’
      In the expression: runRIO () $ logInfo "Hello World!"
      In an equation for ‘main’:
          main = runRIO () $ logInfo "Hello World!"
```

The logging system in `rio` is—hopefully unsurprisingly—built on top of all that `Has*` lens stuff we just finished discussing. It's telling us that the environment we've selected—`()`—does not provide a logging function. Cool. Let's break this in a slightly different way:

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = logInfo "Hello World!"
```

Now the error message is obscured a bit:

```
Main.hs:8:8: error:
    • No instance for (MonadReader env0 IO)
        arising from a use of ‘logInfo’
    • In the expression: logInfo "Hello World!"
      In an equation for ‘main’: main = logInfo "Hello World!"
```

We'll get into understanding this a bit more in the next section on lifting and unlifting. Let's break things in one more way:

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
-- turn this off {-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = runSimpleApp $ logInfo "Hello World!"
```

Now we get the error message:

```
Main.hs:8:31: error:
    • Couldn't match expected type ‘Utf8Builder’
                  with actual type ‘[Char]’
    • In the first argument of ‘logInfo’, namely ‘"Hello World!"’
      In the second argument of ‘($)’, namely ‘logInfo "Hello World!"’
      In the expression: runSimpleApp $ logInfo "Hello World!"
```

Remember how we said that `String` (aka `[Char]`) was an inefficient choice? The logging functions in `rio` agree, and instead use a `Utf8Builder` typeclass. This is built on top of a bytestring `Builder`, and demands that the bytes are UTF-8 encoded. `rio`'s logging system then ensures that the bytes are converted to the appropriate character encoding when sending to the console. For the common case of UTF-8 consoles, this is cheap and efficient, exactly what we want!

In addition to `logInfo`, there are also `logDebug`, `logWarn`, `logError`, and others.

Exercise Guess the output of the following, and then run the program.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = runSimpleApp $ do
  logDebug "Debug"
  logInfo "Info"
  logWarn "Warn"
  logError "Error"
```

By default, `runSimpleApp` will use a non-verbose log function, which does not include debug-level output. So our output is:

```
Info
Warn
Error
```

If we want to get verbose logging with `runSimpleApp`, we can use the `RIO_VERBOSE` environment variable. Here's some example output:

```
$ export RIO_VERBOSE=1
$ ./Main.hs
2019-03-10 10:07:54.721114: [debug] Debug
@(/Users/michael/Desktop/Main.hs:9:3)
2019-03-10 10:07:54.722156: [info] Info
@(/Users/michael/Desktop/Main.hs:10:3)
2019-03-10 10:07:54.722210: [warn] Warn
@(/Users/michael/Desktop/Main.hs:11:3)
2019-03-10 10:07:54.722270: [error] Error
@(/Users/michael/Desktop/Main.hs:12:3)
```

Note that this not only enables debug-level messages, but also prints timestamps, the log level, and the source location where the log message was sent from. Neat!

Bypassing `SimpleApp`

`SimpleApp` is a nice shortcut for writing simple applications. However, you'll often want to have more control of what the log function should be, and will likely want to define your own application environment. So instead of using `runSimpleApp`, let's define our own `App` type. We'll go back to our "say hello" example above.

Exercise Make the code below compile by adding necessary instances. It won't run correctly yet, we'll handle that next.

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

data App = App
  { appLogFunc :: !LogFunc
  , appName :: !Utf8Builder
  }

main :: IO ()
main = runApp sayHello

runApp :: RIO App a -> IO a
runApp = error "we'll do this next"

sayHello :: RIO App ()
sayHello = do
  name <- view $ to appName
  logInfo $ "Hello, " <> name
```

Solution The error message we get with this code is:

```
Main.hs:21:3: error:
    • No instance for (HasLogFunc App) arising from a use of ‘logInfo’
```

If we add just this:

```haskell
instance HasLogFunc App
```

we get the warning:

```
Main.hs:12:10: warning: [-Wmissing-methods]
    • No explicit implementation for
        ‘logFuncL’
    • In the instance declaration for ‘HasLogFunc App’
   |
12 | instance HasLogFunc App
   |          ^^^^^^^^^^^^^^
```

The proper implementation is:

```haskell
instance HasLogFunc App where
  logFuncL = lens appLogFunc (\x y -> x { appLogFunc = y })
```

I told you there'd be some boilerplate and that pattern would keep popping up!

Alright, running this code doesn't work, since we haven't actually implemented `runApp`. Let's get this a small step closer:

```haskell
runApp :: RIO App a -> IO a
runApp inner = do
  let app = App
        { appLogFunc = error "not available"
        , appName = "Alice"
        }
  runRIO app inner
```

But we still don't have a `LogFunc`. We can cheat in two easy ways. First, it turns out that there's a `Monoid` instance for `LogFunc`. This will both compile and run successfully:

```haskell
runApp :: RIO App a -> IO a
runApp inner = do
  let app = App
        { appLogFunc = mempty
        , appName = "Alice"
        }
  runRIO app inner
```

However, the result isn't what we want: there's no output! Not surprising when you consider what an "empty log function" probably looks like. Another way we can cheat is to abuse `runSimpleApp`.

Exercise Get this code to compile:

```haskell
runApp :: RIO App a -> IO a
runApp inner = runSimpleApp $ do
  logFunc <- _
  let app = App
        { appLogFunc = logFunc
        , appName = "Alice"
        }
  runRIO app inner
```

Solution We want to steal the log function from the `SimpleApp` type. We can use:

```haskell
logFunc <- view logFuncL
```

Alright, enough playing around. How do we actually create a log function?!? We go through a two-stage process:

1. Create a `LogOptions` value which defines the options we want for the log function
2. Use `withLogFunc` to create a `LogFunc` from the `LogOptions`

This is similar to what is known as the "builder pattern" in the Java world. The Haddocks describe how to do this. **Exercise** Go ahead and take a stab at implementing `runApp` without cheating with `mempty` or `runSimpleApp`.

Solution Here's a fully working example.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

data App = App
  { appLogFunc :: !LogFunc
  , appName :: !Utf8Builder
  }

instance HasLogFunc App where
  logFuncL = lens appLogFunc (\x y -> x { appLogFunc = y })

main :: IO ()
main = runApp sayHello

runApp :: RIO App a -> IO a
runApp inner = do
  logOptions' <- logOptionsHandle stderr False
  let logOptions = setLogUseTime True $ setLogUseLoc True logOptions'
  withLogFunc logOptions $ \logFunc -> do
    let app = App
          { appLogFunc = logFunc
          , appName = "Alice"
          }
    runRIO app inner

sayHello :: RIO App ()
sayHello = do
  name <- view $ to appName
  logInfo $ "Hello, " <> name
```

Exercise Play around with setting other log settings.

Exercise Generalize `runApp` so that it doesn't mention `IO` in the type signature.

Exercise Modify `sayHello` so that it doesn't mention `App` in the signature. Hint: you'll need to define a new typeclass.

If you've gotten this far and understand what's going on: congratulations! You have enough of a grasp of the `RIO` type to write real world applications with it!

Lifting and unlifting

Let's rewind to an error message we saw way above. The following code:

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = logInfo "Hello World"
```

Produces the error message:

```
Main.hs:8:8: error:
    • No instance for (MonadReader env0 IO)
        arising from a use of ‘logInfo’
    • In the expression: logInfo "Hello World"
      In an equation for ‘main’: main = logInfo "Hello World"
```

From what we've learned so far, this may be a surprising error message. You may have instead expected something like:

```
Main.hs:9:8: error:
    • Couldn't match expected type ‘IO ()’
                  with actual type ‘RIO env0 ()’
```

That's certainly clearer. And that's exactly the error message you would get if `logInfo` had the type signature:

```haskell
logInfo :: HasLogFunc env => Utf8Builder -> RIO env ()
```

But in reality, it doesn't. Instead, the type signature is the more complex:

```haskell
logInfo :: (MonadIO m, MonadReader env m, HasLogFunc env) => Utf8Builder -> m ()
```

(Plus some stuff about `HasCallStack` which we're ignoring.) The latter is a generalization of the former, and allows `logInfo` to be used in monads besides `RIO` itself. This is a design decision in the `rio` library to allow `rio` to be used more generally, in cases where the user isn't fully bought in to the `RIO` data type.

It's always possible to convert from `RIO`-specific functions to more general functions based on `mtl`-style typeclasses like `MonadIO` and `MonadReader`. This is what the `liftRIO` function does:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = runSimpleApp sayHello

sayHelloRIO :: HasLogFunc env => RIO env ()
sayHelloRIO = logInfo "Hello World!"

sayHello :: (MonadReader env m, MonadIO m, HasLogFunc env) => m ()
sayHello = liftRIO sayHelloRIO
```

Going the other way is even easier, since `RIO` is an instance of `MonadIO` and `MonadReader`:

```haskell
sayHelloRIO :: HasLogFunc env => RIO env ()
sayHelloRIO = sayHello

sayHello :: (MonadReader env m, MonadIO m, HasLogFunc env) => m ()
sayHello = logInfo "Hello World!"
```

`liftRIO` and `MonadIO` do not work for all cases. Specifically, they are limited to cases where the `m` appears in *positive position*. If they appear in *negative position*, you'll need to use the `MonadUnliftIO` typeclass. Instead of discussing that in detail here, please see the tutorial on `unliftio`. And in case you're wondering: yes, `RIO` is an instance of `MonadUnliftIO`.

With this in mind: you can use the `rio` library in code even where you're not using the `RIO` data type. The functions all generalize nicely. Our recommendation is to base your code around `RIO`, since the `ReaderT` design pattern bypasses a lot of wasted time. But if you're not bought in, or you've got some other constraints (like legacy code) that force your hand, feel free to use `rio` anyway.

One final point. When should your type signatures use the `RIO` data type, versus using `MonadIO`/`MonadUnliftIO`/`MonadReader`? Our recommendation is to stick to `RIO` unless you know you'll need something more general. Error messages are much nicer with using `RIO` directly, and it's much faster to type. You don't need to get into wasted time thinking about whether you want `MonadIO` or `MonadUnliftIO`. And it's easy enough to convert from `RIO` to the typeclass-based approach (see example below).

One counterexample would be if you're writing a general purpose library that is supposed to support more use cases. That's the case of `rio` itself. In that case, bite the bullet and use the more verbose type signatures.

Unlifting RIO

Getting all of the types correct for unlifting a `RIO` usage can be a bit tricky. As a cookbook example:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO

main :: IO ()
main = pure ()

withLoggedBinaryFileRIO
  :: HasLogFunc env
  => FilePath
  -> IOMode
  -> (Handle -> RIO env a)
  -> RIO env a
withLoggedBinaryFileRIO fp iomode inner = do
  logDebug $ "About to open " <> fromString fp
  withBinaryFile fp iomode $ \h ->
    inner h `finally` logDebug ("Finished using: " <> fromString fp)

withLoggedBinaryFile
  :: (MonadUnliftIO m, MonadReader env m, HasLogFunc env)
  => FilePath
  -> IOMode
  -> (Handle -> m a)
  -> m a
withLoggedBinaryFile fp iomode inner =
  withRunInIO $ \run ->
  run $ liftRIO $ withLoggedBinaryFileRIO fp iomode $ \h ->
  liftIO $ run (inner h)
```

The pattern is:

- Use `withRunInIO` to get a function to convert `m a` actions to `IO a` actions
- Use `run . liftRIO` to convert from a `RIO` action to an `IO` action
- Use `liftIO . run` to convert from an `m a` inner action to a `RIO` action

Not the easiest code in the world, but hopefully not something you'll be doing too often.

Monad transformers

One thing worth pointing out is that, when fully embracing `RIO`, you won't use monad transformers very often. Instead of a transformer, your application will overall live in the `RIO` monad. Instead of, for example, a `MonadLogger` typeclass and `LoggingT` transformers, you have the `HasLogFunc` typeclass. This is all encapsulated in the `ReaderT` design pattern. More information on the advantages of avoiding monad transformers is available in the talk [Everything you didn't want to know about monad transformer state]() ([slides]() also available).

That's not to say that transformers are *never* used in `rio`. In small parts of the codebase, it can be useful to use something like `StateT`, for example. And it's quite common to combine something like `ConduitT` and `RIO`. But for large-scale usage across the whole application: avoid the transformers.

Exception handling

The `rio` library exports exception handling functions. These functions are different from those in `Control.Exception` in two ways:

- They use `MonadIO` and `MonadUnliftIO` instead of hard-coding `IO`

- They have better handling of asynchronous exceptions

Instead of rehashing this information here, please read the exception handling tutorial.

Module hierarchy

The module hierarchy in `rio` is fairly simple, and reading the Stackage page will give you a list of available modules. The structure is:

- Everything lives under `RIO`
- Everything under `RIO.Prelude` is reexported by `RIO` itself
- Partial functions (those which throw exceptions on some input) are exported by `RIO.X.Partial`
- Unsafe functions (those which can segfault on some input) are exported by `RIO.X.Unsafe`
- All modules that should be imported qualified indicate that at the top of the module documentation, and provide a recommended qualified name

Related libraries

`rio` reexports functionality from many other libraries. Instead of including that documentation here, the tutorials on this site use `rio` versions of the libraries wherever possible. So please continue with the documentation at:

- bytestring and text
- containers
- stm
- async
- vector
- typed-process (though see caveat below.md)
- Exception handling
- unliftio
- Mutable variables

When to use `rio`?

Previously, and with other prelude replacements, we've given the recommendation "use in applications, not libraries." That's no longer the case with `rio`. `rio` is a recommended library for all use cases. It's likely that over the next few years, more libraries on Hackage will begin depending on `rio`. Stay tuned!

The `rio` Stack template

There's a Stack template available for `rio` which ties together options parsing, setting up logging and external process running, and a test suite. You can get it with `stack new projectname rio`.

Running external processes

The external process code for `rio` mostly works exactly as the typed-process library. However, there are two important changes:

- There's a `HasProcessContext` typeclass which allows for things like modifying environment variables, caching the results of some expensive operations, and changing the subdirectory.
- The type signature of `proc` is modified to allow for logging how long a process runs for.

The following program demonstrates how to run a program who's arguments are given on the command line.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import RIO
import RIO.Process
import System.Environment
import System.Exit

-- Here comes the boilerplate!
data App = App
  { appLogFunc :: !LogFunc
  , appProcessContext :: !ProcessContext
  }
instance HasLogFunc App where
  logFuncL = lens appLogFunc (\x y -> x { appLogFunc = y })
instance HasProcessContext App where
  processContextL = lens appProcessContext (\x y -> x { appProcessContext = y })

main :: IO ()
main = do
  -- more boilerplate, could use runSimpleApp instead
  lo <- logOptionsHandle stderr True
  pc <- mkDefaultProcessContext
  withLogFunc lo $ \lf ->
    let app = App
          { appLogFunc = lf
          , appProcessContext = pc
          }
     in runRIO app run

run :: RIO App ()
run = do
  args <- liftIO getArgs
  case args of
    [] -> do
      logError "You need to provide a command to run"
      liftIO exitFailure
    x:xs -> proc x xs runProcess_
```

Final exercise

Use `stack new myproject rio` to create a new project. Then modify it to run `git ls-files -z` in a directory specified on the command line, and print out all of the `.md` files.

+1 858-617-0430

sales@fpcomplete.com

Haskell

Training

All Services

Jobs