Safe exception handling

Exception handling can be a bit of a black art in most programming languages with runtime exceptions. Haskell's situation is even more complicated by the presence of *asynchronous exceptions* (described below). On top of that, the functions provided in the `Control.Exception` module make it particularly difficult to get all of the details right.

This tutorial provides instruction on how to do things the right way in Haskell. It's a good idea to understand all of the gory details under the surface as well, though you can get far without those details. There is a blog post, webcast recording, and set of slides available providing an in-depth look at all of this called Async Exception Handling in Haskell.

In this tutorial, we're going to focus on:

- What "safe exception handling" means, at a high level
- Which functions to use
- Common patterns
- Pitfalls to avoid

What we won't do here:

- Cover the full motivation for the design of the libraries in question
- Debate the merits of runtime exceptions
- Debate the merits of asynchronous exceptions

`UnliftIO.Exception`

Instead of using the `Control.Exception` module, we recommend using the `UnliftIO.Exception` module from the `unliftio` package. It provides two important advantages over `Control.Exception`:

- It works in more monads than just `IO` by using the `MonadIO` and `MonadUnliftIO` typeclasses, see the `unliftio` library for more information
- It handles asynchronous exceptions better, as we'll describe below

The contents of the `UnliftIO.Exception` module are reexported from both the `UnliftIO` and `RIO` modules (see the `rio` library). Fo our examples, we're simply going to use `RIO`.

What is safe exception handling?

The definition we're going to use is "all resources are cleaned up promptly despite the presence of an exception." It's easiest to see what safe exception handling is by counterexample. Try to identify what is unsafe here:

```
foo :: IO Result
foo = do
  resource <- openResource
  result <- useResource resource
  closeResource resource
  pure result
```

If `useResource` throws an exception, then `closeResource` will never be called, which would be unsafe resource handling. In reality, we could ensure that the garbage collector cleans up the resources for us, but that fails our "promptly" requirement, since we have no guarantees of when the garbage collector will know it can clean up the resource. For some resources like file descriptors, this can easily cause your entire program to crash.

If you're familiar with most languages with runtime exceptions, you may think that the following is safe:

```
foo :: IO Result
foo = do
  resource <- openResource
  eitherResult <- try $ useResource resource
  closeResource resource
  case eitherResult of
    Left e -> throwIO e
    Right result -> pure result
```

Firstly, the above code won't compile (we'll see why in the next section). However, even if we fix it so that it *does* compile, it's still broken. Haskell's runtime system includes *asynchronous exceptions*. These allow other threads to kill our thread. In the async library, we use this create useful functions like `race`. But in exception handling, these are a real pain. In the code above, an async exception could be receive after the `try` completes but before the `closeResource` call.

Even helpful functions like `finally` aren't sufficient in this case. As an exercise, try to figure out how asynchronous exceptions could cause `closeResource` to not be called in this code:

```
foo :: IO Result
foo = do
  resource <- openResource
  finally
    (useResource resource)
    (closeResource resource)
```

Solution In this case, an asynchronous exception could be received between the call to `openResource` finishing and `finally` beginnir The correct way to use this is to use the `bracket` function (which we'll go into more detail on below):

```
foo :: IO Result
foo = bracket openResource closeResource useResource
```

You can also address this with explicit usage of low level *exception masking* functions. We're explicitly not going to cover that in this tutorial, since it's error prone and rarely needed. Try to stick to the functions we discuss, like `catch` and `bracket`. The in depth blog pos linked above provides the full gory details if desired.

How to throw exceptions

There are three different ways exceptions can be thrown in Haskell:

- Synchronously thrown: an exception is generated from `IO` code and thrown inside a single thread
- Asynchronously thrown: an exception is thrown from one thread to another thread to cause it to terminate early
- Impurely thrown: an exception is generated from pure code, and gets thrown when a thunk is forced

Asynchronous throwing is the odd man out here, so let's ignore it for the moment. When it comes to synchronous throwing, we use the `throwIO` function (or something built on top of it). For example:

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

-- boilerplate, we'll get to this in a bit
data MyException = MyException
  deriving (Show, Typeable)
instance Exception MyException

main :: IO ()
main = runSimpleApp $ do
  logInfo "This will be called"
  throwIO MyException
  logInfo "This will never be called"
```

By contrast, the `impureThrow` function creates a value which, when forced, will throw an exception. For example:

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

-- boilerplate, we'll get to this in a bit
data MyException = MyException
  deriving (Show, Typeable)
instance Exception MyException

main :: IO ()
main = runSimpleApp $ do
  logInfo "This will be called"
  let x = impureThrow MyException
  logInfo "This will also be called"
  if x -- forces evaluation
    then logInfo "This will never be called"
    else logInfo "Neither will this"
```

A common example of impure exceptions you'll see in Haskell code is the `error` function. And in fact, sometimes it even looks and behaves like `throwIO`, such as:

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

main :: IO ()
main = runSimpleApp $ do
  logInfo "This will be called"
  error "Impure or synchronous exception"
  logInfo "Will this be called?"
```

It seems like `error` is the same as `throwIO` here. But it's *ever so slightly* different. What's actually happening is that `error "..."` is receiving the type `RIO SimpleApp ()`. Then that action is forced, which generates a synchronous exception.

The important point for our purposes here: once an impure exception is forced, we treat it as a synchronous exception in every way. Whic brings us to the next bit.

Sync vs async

There's a fundamental difference between how we handle synchronous versus asynchronous exceptions. A sync exception means *something went wrong locally*. We're free to clean up after ourselves, or fully recover. For example, if I try to read a file, and get a "does no exist" exception, it's valid to either rethrow the exception and give up, or to print a warning and continue running with some default value For example:

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

main :: IO ()
main = runSimpleApp $ do
  let fp = "myfile.txt"
  message <- readFileUtf8 fp `catchIO` \e -> do
    logWarn $ "Could not open " <> fromString fp <> ": " <> displayShow e
    pure "This is the default message"
  logInfo $ display message
```

An asynchronous exception is totally different. It is a demand from outside of our control to shut down as soon as possible. If we were to catch such an exception and recover from it, we would be breaking the expectations of the thread that tried to shut us down. Instead, wit asynchronous exceptions, exception handling best practices tell us we're allowed to clean up, but not recover. For example, the `timeout` function uses asynchronous exceptions. What should the expected behavior here be?

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

oneSecond, fiveSeconds :: Int
oneSecond = 1000000
fiveSeconds = 5000000

main :: IO ()
main = runSimpleApp $ do
  res <- timeout oneSecond $ do
    logInfo "Inside the timeout"
    res <- tryAny $ threadDelay fiveSeconds `finally`
      logInfo "Inside the finally"
    logInfo $ "Result: " <> displayShow res
  logInfo $ "After timeout: " <> displayShow res
```

Bad async exception handling would allow the "Result: " message to print. We don't want that to happen! Instead, we allow the `finally` cleanup call to occur and then immediately exit. This ensures that resource cleanup can happen (ensuring exception safety), while disallowing large delays from async exceptions.

In sum, our goals are:

- Synchronous exceptions: allow both recovery and cleanup
- Asynchronous exceptions: allow cleanup, but disallow recovery

We'll see how the functions in `UnliftIO.Exception` fall into these two categories.

Exception types

In addition to how we throw exceptions, there's also the issue of the types of exceptions. This may be surprising, but the Haskell exceptic system is modeled off of Java-style Object Oriented inheritance (shocking, I know). There's a typeclass, `Exception`, and a data type `SomeException` which is the ancestor of all exceptions.

How do you get OO-style inheritance into Haskell? Like this:

```
data SomeException = forall e. Exception e => SomeException e

class (Typeable e, Show e) => Exception e where
  toException :: e -> SomeException
  fromException :: SomeException -> Maybe e
  displayException :: e -> String -- for pretty display purposes
```

Here's how this works: in order for a type to be an exception, it must be possible to convert a value of that type into a `SomeException` value (using the `toException` method). It must also be possible to attempt to convert a `SomeException` into your type from `fromException`, though that conversion may fail. And finally, `SomeException` is nothing more than an existential type saying "I've got something which is a instance of the `Exception` typeclass.

Still confused? Don't worry, that's normal. Let's see an example of defining a simple exception type:

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import Data.Typeable (cast)
import RIO

data MyException = MyException
  deriving (Show, Typeable)
instance Exception MyException where
  -- these are the default implementations, so you can simply omit
  -- them
  toException e = SomeException e
  fromException (SomeException e) = cast e -- uses Typeable

main :: IO ()
main =
  runSimpleApp $
  throwIO MyException `catch` \MyException ->
  logInfo "I caught my own exception!"
```

This uses the `Typeable` typeclass, which allows for runtime type analysis, which is what makes all of this magic work. Love it or hate it, this is at the core of the exception handling mechanism in Haskell.

We can also create hierarchies of exceptions. In my experience, these aren't actually used that often (outside of async exceptions, which we'll get to in a bit).

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ScopedTypeVariables #-}
import Data.Typeable (cast)
import RIO

data Parent = Parent1 Child1 | Parent2 Child2
  deriving (Show, Typeable)
instance Exception Parent

data Child1 = Child1
  deriving (Show, Typeable)
instance Exception Child1 where
  toException = toException . Parent1 -- cast up through the Parent type
  fromException se =
    case fromException se of
      Just (Parent1 c) -> Just c
      _ -> Nothing

data Child2 = Child2
  deriving (Show, Typeable)
instance Exception Child2 where
  toException = toException . Parent2 -- cast up through the Parent type
  fromException se =
    case fromException se of
      Just (Parent2 c) -> Just c
      _ -> Nothing

main :: IO ()
main = runSimpleApp $ do
  throwIO Child1 `catch` (\(_ :: SomeException) -> logInfo "Caught it!")
  throwIO Child1 `catch` (\(_ :: Parent) -> logInfo "Caught it again!")
  throwIO Child1 `catch` (\(_ :: Child1) -> logInfo "One more catch!")
  throwIO Child1 `catch` (\(_ :: Child2) -> logInfo "Missed!")
```

In this case, both `Child1` and `Child2` are children of the `Parent` type, and `Parent` is a child of the `SomeException` type. Therefore, i we throw a `Child1`, catching a `SomeException` or a `Parent` will catch the `Child1`. However, trying to catch a `Child2` will *not* catch the `Child1`, and the exception will escape.

Which brings us to...

Type ambiguity

Why doesn't this compile?

```
foo :: IO ()
foo = do
  resource <- openResource
  eitherResult <- try $ useResource resource
  closeResource resource
  case eitherResult of
    Left e -> throwIO e
    Right result -> pure result
```

The type of `try` is (slightly simplified):

```
try :: Exception e => IO a -> IO (Either e a)
```

Notice the type variable `e`. `try` will catch whichever type of exception you ask it to. But if you're unclear about which exception type you care about, the compiler will complain about ambiguous types. That's why, in our hierarchical exception above, I turned on `ScopedTypeVariables` and included signatures like `` `catch` (\(_ :: Child1) -> ``.

We'll discuss in the recovering section below some common practices around catching exceptions.

Async exceptions

Previously, we pointed out that the difference between a synchronous and asynchronous exception is how they are thrown (via `throwIO` or `throwTo`). Unfortunately, there's no way to determine when catching an exception how it was thrown, making it difficult to live up to o goals above to never recover from an async exception. Fortunately, we have a workaround: use a different type for async exceptions!

Using the hierarchical exception mechanism above, we have a new data type, `SomeAsyncException`, which is a child of `SomeException`. All exceptions which are thrown asynchronously must be a child of that exception type. And conversely, asynchronous exceptions must *not* be thrown synchronously. The `UnliftIO.Exception` module has quite a few safeguards in place to ensure both c these conditions are met. Please see the "Async Exception Handling in Haskell" article above for the gory details.

Upshot of all of this:

- If you define your own exception type for asynchronous exceptions, make it a child of `SomeAsyncException`. (Note: this is a pret unusual thing to do.)
- If you define your own exception type for synchronous exceptions, don't make it a child of `SomeAsyncException`.
- The functions we'll mention below for cleaning up and recovering are able to determine whether an exception is sync or async base on its type.

Cool? Awesome! That's quite enough backstory. Let's start covering usage of the API.

Throwing

We're not going to talk about using async exceptions to kill other threads, since we're not talking about concurrent programming. Instead please check out the async library and the `race` and `cancel` functions it provides. Instead, we're going to focus on synchronous exception throwing.

The most basic function for this is:

```
throwIO :: (MonadIO m, Exception e) => e -> m a
```

Given any value which is an instance of `Exception`, you can throw it as a runtime exception for any monad which is a `MonadIO` instance

This works for built in exception types, as well as any you define yourself.

Sometimes you want to use synchronous exceptions but don't want to go through the overhead of defining your own exception type. In those cases, you can use the helper:

```
throwString :: (MonadIO m, HasCallStack) => String -> m a
```

`throwString` looks pretty similar to `error`:

```
error :: HasCallStack => String -> a
```

The difference is that the former throws a synchronous exception of type `StringException`, whereas the latter creates a thunk which, when evaluated, throws a synchronous exception of type `ErrorCall`. To demonstrate the difference:

```
throwString "foo" :: IO () -- throws an exception
error "foo" :: IO () -- throws an exception, because the thunk is evaluated
throwString "foo" `seq` pure () :: IO () -- doesn't throw an exception!
error "foo" `seq` pure () :: IO () -- does throw an exception!
throwString "foo" :: () -- type error
error "foo" :: () -- compiles, and when evaluated will throw an exception
```

Typically, we advise away from exceptions in pure code, and thus `error` is best avoided. But *if* you really want an exception in pure code you can also use `impureThrow`, which lets you use your own exception type:

```
impureThrow :: Exception e => e -> a
```

Cleaning up

Cleaning up allows you to define some action which should be run when an exception occurs. However, after your action is run, the exception will be rethrown. In other words, when cleaning up, you *cannot recover* from an exception. This makes cleanup functions safe t use with asynchronous exceptions.

The simplest cleanup function is `finally`, which ensures that an action is run whether or not an exception is thrown:

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

main :: IO ()
main = runSimpleApp $ do
  logInfo "This will print first"
  throwString "This will print last as an error message"
    `finally` logInfo "This will print second"
  logInfo "This will never print"
```

Similar is the `onException` function, which will only run its second argument if the first argument exited with an exception.

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

main :: IO ()
main = runSimpleApp $ do
  logInfo "This will print first"
    `onException` logInfo "This will never print"
  throwString "This will print last as an error message"
    `onException` logInfo "But this will print second"
  logInfo "This will never print"
```

And the most commonly used of this cleanup functions is likely `bracket`. `bracket` is so popular that there's even a style of functions called "the bracket pattern." `bracket` takes a resource allocation function, a resource cleanup function, and a function to use the resourc and ensures that cleanup occurs. It looks like:

```
bracket
  :: MonadUnliftIO m
  => m a -- ^ allocate
  -> (a -> m b) -- ^ cleanup
  -> (a -> m c) -- ^ use
  -> m c
```

Exercise Implement a `withBinaryFile` function (specialized to `IO` for simplicity) using `bracket`, `hClose`, and `System.IO.openBinaryFile`.

There are a few other functions available, like `withException`. Overall, these functions are fairly easy to understand, but take some experience to know how to use correctly.

Recovering

The exception recovery functions allow you to catch an exception and prevent it from propagating higher up the call stack. These functions only work on synchronous exceptions; they will totally ignore asynchronous exceptions. We'll start with the `try` family of functions, and then introduce the very similar `catch` and `handle` families.

The basic function is `try`:

```
try :: (MonadUnliftIO m, Exception e) => m a -> m (Either e a)
```

If the provided action was successful, it will return a `Right`, otherwise it will return a `Left`. And if the provided action throws a different type of exception than expected, that exception will be rethrown.

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

main :: IO ()
main = runSimpleApp $ do
  res1 <- try $ throwString "This will be caught"
  logInfo $ displayShow (res1 :: Either StringException ())

  res2 <- try $ pure ()
  logInfo $ displayShow (res2 :: Either StringException ())

  res3 <- try $ throwString "This will be caught"
  logInfo $ displayShow (res3 :: Either SomeException ())

  res4 <- try $ throwString "This will *not* be caught"
  logInfo $ displayShow (res4 :: Either IOException ())
```

Having to specify the type of exception you want can be tedious, so there are two helper functions that address common cases. The `tryIO` function is specialized to `IOException`, which is the type used by many function in standard libraries which perform I/O. For example:

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

main :: IO ()
main = runSimpleApp $ do
  result <- tryIO $ readFileUtf8 "does-not-exist"
  case result of
    Left e -> logError $ "Error reading file: " <> displayShow e
    Right text -> logInfo $ "That's surprising... " <> display text
```

Notice how, in the above, we didn't need any explicit type signatures. The type of `e` is constrained to be `IOException`.

The other helper function is `tryAny`, which constraints the exception to be `SomeException`. As we discussed above, `SomeException` the parent exception type in the hierarchy, and therefore this function will catch *all* synchronous exceptions. We can replace `tryIO` with `tryAny` above, and the error message will remain unchanged.

One more aspect of the `try` family: remember those pesky impure exceptions? Well, it's possible for them to leak through. For example:

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

main :: IO ()
main = runSimpleApp $ do
  result1 <- tryAny $ error "This will be caught"
  case result1 of
    Left _ -> logInfo "Exception was caught"
    Right () -> logInfo "How was this successful?!?"

  result2 <- tryAny $ pure $ error "This will escape!"
  case result2 of
    Left _ -> logInfo "Exception was caught"
    Right () -> logInfo "How was this successful?!?"
```

In the first case, the impure exception was forced, turned into a synchronous exception, and caught by `tryAny`. In the second case, however, we wrapped up the impure exception with `pure`, preventing it from being forced. As far as `tryAny` is concerned, the second action succeeded! Then, when we try to pattern match on `result2`, we force the impure exception hiding inside the `Right`.

To work around this, we have the `tryAnyDeep` function, which forces the value using `NFData`. Swapping out `tryAny` with `tryAnyDeep` will result in both exceptions being caught.

Once you understand how to use `try`, using `catch` and `handle` is basically the same thing. Instead of returning an `Either` value, you provide these functions with actions to perform with the exception if one occurs. The type signatures are:

```
catch :: (MonadUnliftIO m, Exception e) => m a -> (e -> m a) -> m a
handle :: (MonadUnliftIO m, Exception e) => (e -> m a) -> m a -> m a
```

`handle` is just `catch` with the order of arguments reversed. These functions come with all the variations of `try` mentioned aboved (e.g. `catchIO`, `handleAny`). Feel free to use whichever is the most convenient.

Exercise Implement `catch` and `handle` in terms of `try`, and implement `try` in terms of `catch`.

Exceptions best practices

NOTE: This was originally in a separate blog post.

Now that you understand how to work with exceptions, let's give some guidelines on best practices in Haskell. This is an opinionated set guidelines. It ties in closely with our recommended approach in the `rio` library.

- The Haskell runtime system allows any `IO` action to throw runtime exceptions. Many common library functions throw runtime exceptions. There is no indication at the type level if something throws an exception. You should assume that, unless explicitly documented otherwise, all actions may throw an exception.
- It's not worth trying to fight the system and make all possible error cases explicit. Other languages, like Rust, go this route, and it works well in practice. But that's because it's an ethos throughout the entire ecosystem. In Haskell, you're fighting against the current.
- As a result: making large sum types containing all possible exceptions, and wrapping all your `IO` code in `ExceptT`, is a major anti-pattern. It has three major downsides:

    - It slows down your code.
    - It introduces confusion: your type signatures imply that it has only one way to fail (the explicit error case), but in reality runtin exceptions still exist.
    - It becomes much harder to use combinators like `concurrently` safely due to monadic state.
- Exception to the above rule: it can sometimes be convenient for small blocks of code to use `ExceptT` or `MaybeT` to avoid deeply nested code blocks.
- Avoid explicit masking of exceptions like the plague. It is fraught with disaster. Use higher level combinators. And do not attempt tc "fix" async exceptions by masking your entire code base. Again: you're fighting against the Haskell ecosystem!
- Use custom exception types with nice `displayException` implementations whenever possible.
- Don't be afraid to throw a runtime exception from your code if something exceptional happens. Ideally, document this case.
- On the other hand, if a failure is not really exceptional, represent it in the return type. For example: a `lookup` in a `Map` shouldn't throw a runtime exception, it should return a `Maybe` value.
- The line between exceptional and non-exceptional is pretty blurry in many cases, so use your best judgement based on your domai

Going deeper

This is a relatively high level overview of exception handling in Haskell. As mentioned, the trick is to get a lot of practice using the functions above. And if you're interested in getting a much deeper intuition, please check out Async Exception Handling in Haskell.

Signup for our Haskell mailing list