-
- Parallel and Concurrent Programming in Haskell

- Comments Off
- Chapters

Table of Contents

Chapter 8. Overlapping Input/Output

We can use `MVar` and threads to do asynchronous I/O, where "asynchronous" in this context means that the I/O is performed in the background while we do other tasks.

Suppose we want to download some web pages concurrently and wait for them all to download before continuing. We will use the following function to download a web page:

```
getURL :: String -> IO ByteString
```

This function is provided by the module `GetURL` in `GetURL.hs`, which is a small wrapper around the API provided by the `HTTP` package.

Let's use `forkIO` and `MVar` to download two web pages at the same time:

geturls1.hs

```
import Control.Concurrent
import Data.ByteString as B
import GetURL

main = do
  m1 <- newEmptyMVar                                    -- 1
  m2 <- newEmptyMVar                                    -- 2

  forkIO $ do                                           -- 3
    r <- getURL "http://www.wikipedia.org/wiki/Shovel"
    putMVar m1 r

  forkIO $ do                                           -- 4
    r <- getURL "http://www.wikipedia.org/wiki/Spade"
    putMVar m2 r

  r1 <- takeMVar m1                                     -- 5
  r2 <- takeMVar m2                                     -- 6
  print (B.length r1, B.length r2)                      -- 7
```

1 Create two new empty `MVar`s to hold the results.

2

3 Fork a new thread to download the first URL; when the download is complete, the result is placed in the `MVar` `m1`.

4. Do the same for the second URL, placing the result in `m2`.

5. In the main thread, this call to `takeMVar` waits for the result from `m1`.

6. Similarly, wait for the result from `m2` (we could do these in either order).

7. Finally, print out the length in bytes of each downloaded page.

This code is rather verbose. We could shorten it by using various existing higher-order combinators from the Haskell library, but a better approach would be to extract the common pattern as a new abstraction. We want a way to perform an action *asynchronously* and later wait for its result. So let's define an interface that does that, using `forkIO` and `MVar`:

```
data Async a = Async (MVar a)

async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyMVar
  forkIO (do r <- action; putMVar var r)
  return (Async var)

wait :: Async a -> IO a
wait (Async var) = readMVar var
```

First, we define an `Async` data type that represents an asynchronous action that has been started. Its implementation is just an `MVar` that will contain the result. Again, we are creating a new data type so as to hide implementation details from clients, and indeed later in this chapter we will need to extend the `Async` type with more information.

It is important to use `readMVar` in `wait`, because this allows multiple `wait` calls to be made for the same `Async`. If we had used a simple `takeMVar`, the second and subsequent calls to `wait` would deadlock. Multiple calls to `wait` for the same `Async` might arise if we are programming in a dataflow style, as in a program that creates a single `Async` and then two further `Async`s that both wait for the result of the first one. In this sense, `Async` is behaving rather like `IVar` from the `Par` monad (Chapter 4), although here, the individual operations are side-effecting `IO` operations rather than pure computations and there is no guarantee of determinism.

Now we can use the `Async` interface to clean up our web page downloading example:

geturls2.hs

```
main = do
  a1 <- async (getURL "http://www.wikipedia.org/wiki/Shovel")
  a2 <- async (getURL "http://www.wikipedia.org/wiki/Spade")
  r1 <- wait a1
  r2 <- wait a2
  print (B.length r1, B.length r2)
```

Much nicer! To elaborate upon this slightly, we can make a small wrapper called `timeDownload` that downloads a URL and reports how much data was downloaded and how long it took, and then apply this to a list of URLs using `async`:

geturls3.hs

```
sites = ["http://www.google.com",
         "http://www.bing.com",
         "http://www.yahoo.com",
         "http://www.wikipedia.com/wiki/Spade",
         "http://www.wikipedia.com/wiki/Shovel"]

timeDownload :: String -> IO ()
timeDownload url = do
  (page, time) <- timeit $ getURL url   -- 1
  printf "downloaded: %s (%d bytes, %.2fs)\n" url (B.length page) time

main = do
 as <- mapM (async . timeDownload) sites  -- 2
 mapM_ wait as                            -- 3
```

1. To time the `getURL` call, we use an auxiliary function `timeit` (defined in *TimeIt.hs*).

2. `mapM` maps a function over a list in a monad; in this case, the `IO` monad. The function we are mapping over the list is the composition of `async` and `timeDownload`. That is, for each URL in the list, we will create an `Async` that calls `timeDownload` for that URL. The result of the `mapM` call is the list of `Async`s created, which we bind to `as`.

3. Then we `wait` for each of the `Async`s to complete. Notice that in this example, each `Async` is returning only a `()` token when it completes, rather than the web page contents as in the earlier examples. Hence we're using `mapM_`, a variant of `mapM` that ignores the result of applying the function to each list element and returns `()`.

The program produces output like this:

```
downloaded: http://www.google.com (14524 bytes, 0.17s)
```

```
downloaded: http://www.bing.com (24740 bytes, 0.18s)
downloaded: http://www.wikipedia.com/wiki/Spade (62586 bytes, 0.60s)
downloaded: http://www.wikipedia.com/wiki/Shovel (68897 bytes, 0.60s)
downloaded: http://www.yahoo.com (153065 bytes, 1.11s)
```

Our little `Async` API captures a common pattern that occurs with concurrent programming, but so far we have ignored one crucial detail: error handling. To deal with errors, we will need to understand how exceptions work in Haskell, and so the next section will review Haskell's exception-handling support before we return to the question of error handling in "Error Handling with Async".

Exceptions in Haskell

The Haskell 98 and 2010 standards provide a limited form of exceptions in the `IO` monad. The `IO` exception mechanism has been extended by the `Control.Exception` module that comes with GHC to include exceptions generated by purely functional code (e.g., `error` and pattern-matching failure), and to define an extensible hierarchy of exception types. The result of this incremental development is that there are some inconsistencies in the APIs as the Haskell 98/2010 interfaces are gradually replaced by the new, more general APIs.

Haskell has no special syntax or built-in semantics for exception handling; everything is done with library functions. Thus, the idioms for exception catching in particular may look a little strange. The tradeoff is that we are able to build higher-level exception handling combinators that embody more powerful abstractions, as we shall see shortly.

In Haskell, exceptions are thrown by the `throw` function:

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

Two things to note here:

- `throw` takes a value of any type that is an instance of the `Exception` type class.
- `throw` returns the unrestricted type variable `a`, so it can be called from anywhere.

The `Exception` type class is provided by the `Control.Exception` module and is defined as follows:

```
class (Typeable e, Show e) => Exception e where
  -- ...
```

Its methods are not important here (see the documentation for details), but the important principle is that any type that is an instance of both `Typeable` and `Show` can be an `Exception`.[32]

One common type used as an exception is `ErrorCall`:

```
newtype ErrorCall = ErrorCall String
    deriving (Typeable)

instance Show ErrorCall where { ... }

instance Exception ErrorCall
```

For example, we can throw an `ErrorCall` like so:

```
  throw (ErrorCall "oops!")
```

In fact, the function `error` from the `Prelude` does exactly this and is defined as:

```
error :: String -> a
error s = throw (ErrorCall s)
```

I/O operations in Haskell also throw exceptions to indicate errors, and these are usually values of the `IOException` type. Operations to build and inspect `IOException` can be found in the `System.IO.Error` library.

Exceptions in Haskell can be caught, but *only in the `IO` monad*. The basic exception-catching function is `catch`:

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

The `catch` function takes two arguments:

- The `IO` operation to perform, of type `IO a`
- An exception handler of type `e -> IO a`, where `e` must be an instance of the `Exception` class

The behavior is as follows: the `IO` operation in the first argument is performed, and if it throws an exception *of the type expected by the handler*, `catch` executes the handler, passing it the exception value that was thrown. So a call to `catch` catches only exceptions of a particular type, determined by the argument type of the exception handler.

To demonstrate this, we will need a new exception type. Let's make our own in GHCi.[33] First some setup:

```
> import Prelude hiding (catch) -- not needed for GHC 7.6.1 and later
> import Control.Exception
> import Data.Typeable
> :set -XDeriveDataTypeable
```

Remember that to make a type an instance of `Exception`, it must also be an instance of `Show` and `Typeable`. To enable automatic derivation for `Typeable`, we need to turn on the `-XDeriveDataTypeable` flag.

In GHC 7.4.x and earlier, the `Prelude` exports a function called `catch`, which is similar to `Control.Exception.catch` but restricted to `IOException`s. If you're using exceptions with GHC 7.4.x or earlier, you should use the following:

```
import Control.Exception
import Prelude hiding (catch)
```

Note that this code still works with GHC 7.6.1 and later, because it is now a warning, rather than an error, to mention a nonexistent identifier in a `hiding` clause.

Now we define a new type and make it an instance of `Exception`:

```
> data MyException = MyException deriving (Show, Typeable)
> instance Exception MyException
```

Then we check that we can `throw` it:

```
> throw MyException
*** Exception: MyException
```

OK, now to catch it. The `catch` function is normally used infix, like this: *`action` `catch` `\e -> handler`*.

If we try to call `catch` without adding any information about the type of exception to catch, we will get an ambiguous type error from GHCi:

```
> throw MyException `catch` \e -> print e

<interactive>:10:33:
    Ambiguous type variable `a0' in the constraints:
      (Show a0) arising from a use of `print' at <interactive>:10:33-37
      (Exception a0)
        arising from a use of `catch' at <interactive>:10:19-25
    Probable fix: add a type signature that fixes these type variable(s)
    In the expression: print e
    In the second argument of `catch', namely `\ e -> print e'
    In the expression: throw MyException `catch` \ e -> print e
```

So we need to add an extra type signature to tell GHCi which type of exceptions we wanted to catch:

```
> throw MyException `catch` \e -> print (e :: MyException)
MyException
```

The exception was successfully thrown, caught by the `catch` function, and printed by the exception handler. If we throw a different type of exception, it won’t be caught by this handler:

```
> throw (ErrorCall "oops") `catch` \e -> print (e :: MyException)
*** Exception: oops
```

What if we wanted to catch *any* exception? In fact, it is possible to do this because the exception types form a hierarchy, and at the top of the hierarchy is a type called `SomeException` that includes all exception types. Therefore, to catch any exception, we can write an exception handler that catches the `SomeException` type:

```
> throw (ErrorCall "oops") `catch` \e -> print (e :: SomeException)
oops
```

Writing an exception handler that catches all exceptions is useful in only a couple of cases, though:

- Testing and debugging, as in the above example
- Performing some cleanup, before re-throwing the exception

Catching `SomeException` and then continuing is not good practice in production code, because for obvious reasons it isn't a good idea to ignore unknown error conditions.

The `catch` function is not the only way to catch exceptions. Sometimes it is more convenient to use the `try` variant instead:

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

For example:

```
> try (readFile "nonexistent") :: IO (Either IOException String)
Left nonexistent: openFile: does not exist (No such file or directory)
```

Another variant of `catch` is `handle`, which is just `catch` with its arguments reversed:

```
handle :: Exception e => (e -> IO a) -> IO a -> IO a
```

This is particularly useful when the exception handler is short but the action is long. In this case, we can use a pattern like this:

```
  handle (\e -> ...) $ do
    ...
```

It is often useful to be able to perform some operation if an exception is raised and then re-throw the exception. For this, the `onException` function is provided:

```
onException :: IO a -> IO b -> IO a
```

This is straightforwardly defined using `catch`:

```
onException io what
   = io `catch` \e -> do _ <- what
                         throwIO (e :: SomeException)
```

To re-throw the exception here we used `throwIO`, which is a variant of `throw` for use in the `IO` monad:

```
throwIO :: Exception e => e -> IO a
```

It is always better to use `throwIO` rather than `throw` in the `IO` monad because `throwIO` guarantees strict ordering with respect to other `IO` operations, whereas `throw` does not.

We end this short introduction to exceptions in Haskell with two very useful functions, `bracket` and `finally`:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c

finally :: IO a -> IO b -> IO a
```

These are two of the higher-level abstractions mentioned earlier. The `bracket` function allows us to set up an exception handler to reliably deallocate a resource or perform some cleanup operation. For example, suppose we want to create a temporary file on the file system, perform some operation on it, and have the temporary file reliably removed afterward—even if an exception occurred during the operation. We could use `bracket` like so:

```
  bracket (newTempFile "temp")
          (\file -> removeFile file)
          (\file -> ...)
```

In a call `bracket a b c`, the first argument `a` is the operation that allocates the resource (in this case, creating the temporary file), the second argument `b` deallocates the resource again (in this case, deleting the temporary file), and the third argument `c` is the operation to perform. Both `b` and `c` take the result of `a` as an argument. In this case, that means they have access to the name of the temporary file that was created.

The `bracket` function is readily defined using the pieces we already have:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after during = do
  a <- before
  c <- during a `onException` after a
  after a
  return c
```

This definition suffices for now, but note that later in Chapter 9, we will revise it to add safety in the presence of thread cancellation.

The `finally` function is a special case of `bracket`:

```
finally :: IO a -> IO b -> IO a
finally io after = do
  io `onException` after
  after
```

Again, we will be revising this definition later.

Error Handling with Async

If we run the `geturls2` example with the network cable unplugged, we see something like this:

```
$ ./geturls2
geturls2: connect: does not exist (No route to host)
geturls2: connect: does not exist (No route to host)
geturls2: thread blocked indefinitely in an MVar operation
```

What happens is that the two calls to `getURL` fail with an exception, as they should. This exception propagates to the top of the thread that `async` created, where it is caught by the default exception handler that every `forkIO` thread gets. The default exception handler prints the exception to `stderr`, and then the thread terminates. So in `geturls2`, we see two network errors printed. But now, because these threads have not called `putMVar` to pass a result back to the main thread, the main thread is still blocked in `takeMVar`. When the child threads exit after printing their error messages, the main thread is then deadlocked. The runtime system notices this and sends it the `BlockedIndefinitelyOnMVar` exception, which leads to the third error message, shown earlier.

This explains what we saw, but clearly this behavior is not what we want: the program is deadlocked after the error rather than exiting gracefully or handling it. The natural behavior would be for the error to be made available to the thread that calls `wait` because that way the caller can find out whether the asynchronous computation returned an error or a result and act accordingly. Moreover, a particularly convenient behavior is for `wait` to simply propagate the exception in the current thread so that in the common case the programmer need not write any error-handling code at all.

To implement this, we need to elaborate on `Async` slightly:

geturls4.hs

```
data Async a = Async (MVar (Either SomeException a)) -- (1)

async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyMVar
  forkIO (do r <- try action; putMVar var r)  -- (2)
  return (Async var)

waitCatch :: Async a -> IO (Either SomeException a) -- (3)
waitCatch (Async var) = readMVar var

wait :: Async a -> IO a -- (4)
wait a = do
  r <- waitCatch a
  case r of
    Left e  -> throwIO e
    Right a -> return a
```

(1) Where previously we had `MVar a`, now we have `MVar (Either SomeException a)`. If the `MVar` contains `Right a`, then the operation completed successfully and returned value `a`; whereas if it contains `Left e`, then the operation threw the

exception `e`.

2. The `action` is now wrapped in `try`, which returns `Either SomeException a`—exactly the type we need to put into the `MVar`. Earlier, we cautioned that catching `SomeException` is often not a good idea, but this is one case where it is fine because we are catching exceptions in one thread with the intention of propagating them to another thread, and we want the behavior to be the same for *all* exceptions.

3. Now we will provide two ways to wait for the result of an `Async`. The first, `waitCatch`, returns `Either SomeException a` so the caller can handle the error immediately.

4. The second way to wait for a result is `wait`, which has the same type as before. However, now, if `wait` finds that the `Async` resulted in an exception, the exception is re-thrown by `wait` itself. This implements the convenient error-propagating behavior mentioned previously.

Using this new `Async` layer, our `geturls` example now fails more gracefully (see *geturls4.hs* for the complete code):

```
$ ./geturls4
geturls4: connect: timeout (Connection timed out)
[3]    25198 exit 1     ./geturls4
$
```

The program exited with an error code after the first failure, rather than deadlocking as before.

The basic `Async` API is the same as before—`async` and `wait` have the same types—but now it has error-handling built in, and it is much harder for the programmer to accidentally forget to handle errors. The only way to ignore an error is to ignore the result as well.

Merging

Suppose we want to wait for one of several different events to occur. For example, when downloading multiple URLs, we want to perform some action as soon as the first one has downloaded.

The pattern for doing this with `MVar` is that each of the separate actions must put its results into the same `MVar`, so that we can then call `takeMVar` to wait for the first such event to occur. Here is the *geturls3.hs* example from Chapter 8, modified to wait for the *first* URL to complete downloading and then to report which one

it was.

geturls5.hs

```
sites = ["http://www.google.com",
         "http://www.bing.com",
         "http://www.yahoo.com",
         "http://www.wikipedia.com/wiki/Spade",
         "http://www.wikipedia.com/wiki/Shovel"]

main :: IO ()
main = do
  m <- newEmptyMVar
  let
    download url = do
       r <- getURL url
       putMVar m (url, r)

  mapM_ (forkIO . download) sites

  (url, r) <- takeMVar m
  printf "%s was first (%d bytes)\n" url (B.length r)
  replicateM_ (length sites - 1) (takeMVar m)
```

Here, we create a single `MVar` and then fork a thread for each of the URLs to download. Each thread writes its result into the same `MVar`, where the result is now a pair of the URL and its contents. The main thread takes the first result from the `MVar`, announces which URL was the quickest to download, and then waits for the rest of the results to arrive.

```
$ ./geturls5
http://www.google.com was first (10483 bytes)
$
```

While this pattern works, it can be a little inconvenient to arrange it so that all the events feed into the same `MVar`. For example, suppose we want to extend our `Async` API to allow waiting for either of two `Async`s simultaneously, returning the result of the first one to succeed or propagating the exception if either `Async` fails. The function we want is `waitEither`, with this type:

```
waitEither :: Async a -> Async b -> Async (Either a b)
```

Note that because the input `Async`s have already been created, we are too late to tell them to put their results into the same `MVar`. Instead, we have to create two *new* threads to collect the results of each `Async` and merge them into a new `MVar`:

geturls6.hs

```
waitEither :: Async a -> Async b -> IO (Either a b)
waitEither a b = do
  m <- newEmptyMVar
  forkIO $ do r <- try (fmap Left  (wait a)); putMVar m r
  forkIO $ do r <- try (fmap Right (wait b)); putMVar m r
  wait (Async m)
```

To get the right error-handling behavior, `waitEither` uses `wait` to grab each result wrapped in a `try` to catch any exceptions and then puts each result into the newly created `MVar` `m`. Then we make a new `Async` from `m` and `wait` for the result of that.

We can generalize `waitEither` to wait for a list of `Async`s, returning the result from the first one to complete:

```
waitAny :: [Async a] -> IO a
waitAny as = do
  m <- newEmptyMVar
  let forkwait a = forkIO $ do r <- try (wait a); putMVar m r
  mapM_ forkwait as
  wait (Async m)
```

Now, `waitAny` can be used to rewrite *geturls5.hs* using `Async`:

geturls6.hs

```
main :: IO ()
main = do
  let
    download url = do
       r <- getURL url
       return (url, r)

  as <- mapM (async . download) sites

  (url, r) <- waitAny as
  printf "%s was first (%d bytes)\n" url (B.length r)
  mapM_ wait as
```

The code for `waitAny` is quite short and does the job, but it is slightly annoying to have to create an extra thread per `Async` for this simple operation. Threads might be cheap, but we ought to be able to merge multiple sources of events more directly. Later in Chapter 10, we will see how software transactional memory allows a neater and more efficient implementation of `waitAny`.

---

[32] An introduction to `Typeable` is beyond the scope of this book; please refer to the documentation for the module `Data.Typeable`.

[33] For this example to work in GHCi, you will need at least GHC 7.4.1

- Terms of Service
- Privacy Policy
- Interested in sponsoring content?