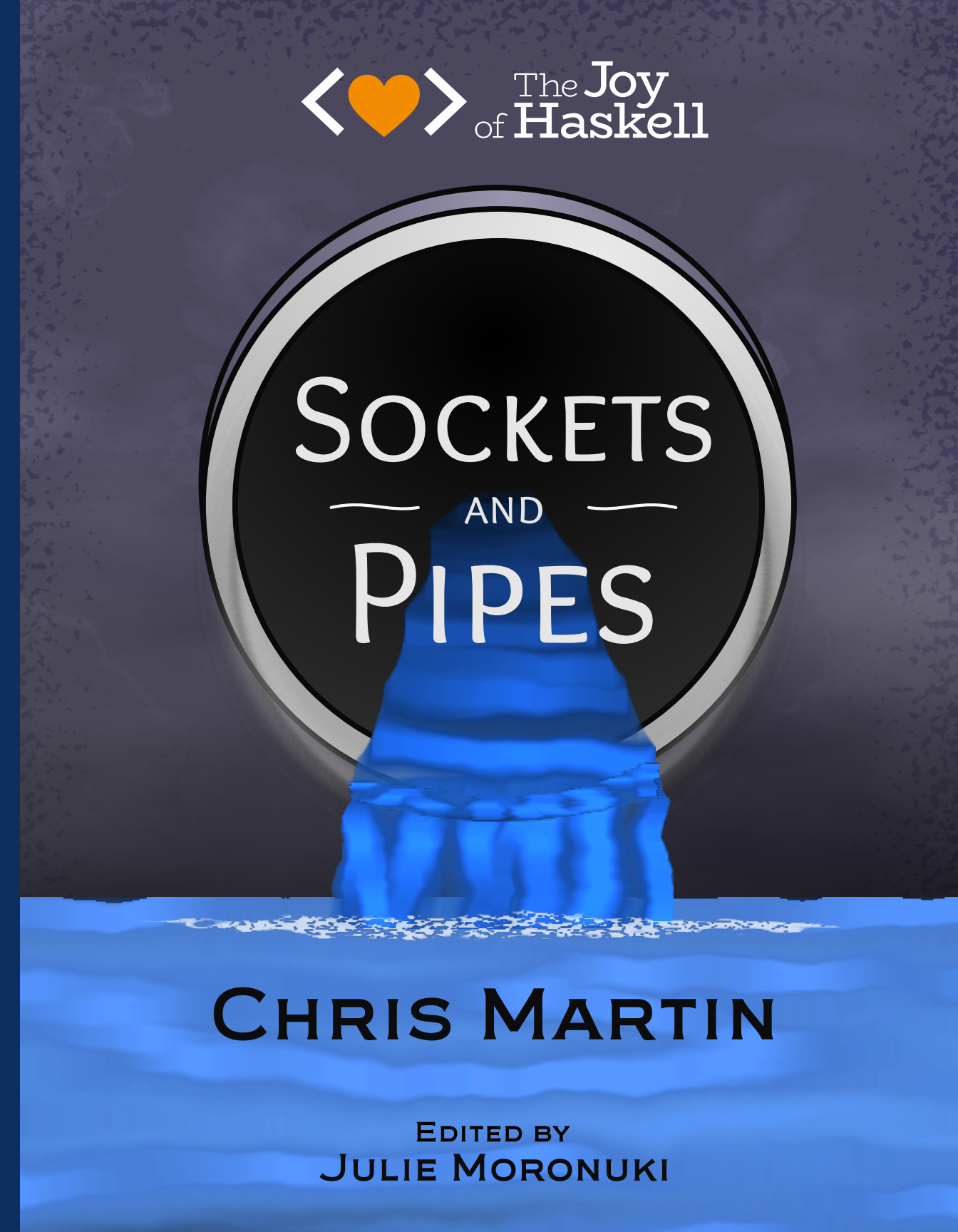
The Joy of Haskell
Sockets and Pipes
Chris Martin
Edited by
Julie Moronuki

Sockets and Pipes
by Chris Martin; edited by Julie Moronuki

2023-05-09: First edition

Contents

Preface

The content that eventually grew into this book began with the question: What exactly *is* a web server? A satisfactory answer that does not assume substantial background knowledge requires spanning quite a few areas of computing. Fortunately, they all serve as fruitful motivations for simultaneously learning about how to use Haskell, which is the larger objective of the *Joy of Haskell* collection.

The language a web server speaks is the Hypertext Transfer Protocol (HTTP), which this book explores in great detail while walking through the creation of a server from "scratch." We encourage readers to follow along in reading the official definition of HTTP (RFC 9110 and 9112 published by the Internet Engineering Task Force) as we implement the specification in Haskell. While high-level libraries make it possible to create web applications without detailed knowledge of HTTP, we believe that a full understanding of the underlying layers we build upon helps us use a platform more effectively. By studying HTTP we also gain an appreciation for what it is and is not good for, and for what applications we might stand to benefit from choosing a different network protocol instead.

Prerequisites

This book is for Haskell learners who have some basic faculty with the language and are now ready to work up to a substantial project. We expect that you understand the basic syntax and can do things like install GHC, use GHCi, write a `case` expression, sequence `IO` actions in a `do` block, use qualified imports, and define datatypes. From the `base` package, we assume some familiarity with `Maybe`, `Either`, `[]`, `Eq`, `Show`, `Monoid`, `Foldable`, `Functor`, and `Monad`. We do not assume prior knowledge of any additional libraries or GHC language extensions.

What's inside

Bytes and characters The first several chapters introduce the `bytestring` and `text` libraries and are largely dedicated to tearing apart a traditional *hello world* program, looking under-

neath the abstract notion of "printing text" to start greeting the world in terms of writing bytes to a file handle. After discussing bytes, we need only a short hop to *sockets*, our means of writing bytes across great distances using the **network** library.

Encoding and parsers First we encode HTTP messages as byte strings. That's the easy part; next, we go in the opposite direction and learn how to interpret byte strings using the **attoparsec** library. This will acquaint us even more closely with the HTTP message format.

Monad transformers We introduce some monad transformers that are especially applicable to our subject matter: **ResourceT**, **Producer**, **ExceptT**, and **ReaderT**. No prior experience with transformers is required. We do not linger on the general concept, preferring instead to focus on each of the three examples and to create familiarity with transformers and lifting chiefly by demonstration. In chapter 16 we begin using **newtype** wrappers and **DerivingVia** to manage the complexity of larger monad stacks.

Resource safety Use of **ResourceT** begins in chapter 1, and we use it throughout the book. This makes it a breeze to deal with files and sockets without resource leaks.

Streaming To move past toy examples that fit easily into memory, we have to start writing *streaming* processes that can deal with large amounts of data by handling it in smaller pieces. All of the code within this book is written with memory usage in mind. **Producer**, **Pipe**, and **Consumer**, the subject of chapter 12, constitute an especially convenient facility for working with streams.

Error handling As features accumulate, the number of possible error conditions starts to rise. Chapter 14 introduces **ExceptT** to work with errors in a clean and well-typed manner.

On Hackage

This book has a companion Haskell library called **sockets-and-pipes**, available from the standard package repository.

https://hackage.haskell.org/package/sockets-and-pipes

The library re-exports all of the modules from other libraries that we use in the book; prospective readers are encouraged to browse the documentation at the web address given above, as it provides an overview of the libraries that you will learn to use from reading this book.

Chapter 0

Setup

We strongly encourage you to follow along with the book and type the code as you read. The exercises at the end of each chapter make use of the code given in the chapter. Subsequent chapters will also refer back to definitions from earlier in the book, so it is important to keep everything as you progress.

GHC version 9.2 or 9.4 is required. You can organize the code however you like, but here we give a recommended setup for the convenience of less opinionated readers.

book.cabal

```
cabal-version: 3.0
name: book
version: 0.0.0.0

common base
    default-language: GHC2021
    default-extensions: BlockArguments DerivingVia NoImplicitPrelude QuasiQuotes
    ghc-options: -Wall -fdefer-typed-holes
    build-depends: sockets-and-pipes ^>= 1.0

library
    import: base
    hs-source-dirs: library
    exposed-modules: Book
```

cabal-version, **name**, and **version** are necessities in any Cabal package file.

The **default-extensions** field enables a few language extensions:

- *Block arguments* is a small adjustment to the Haskell syntax which allows a **do** block to be used as a function parameter, a task which has traditionally been accomplished using the **($)** operator.
- *Deriving via* will be used in chapter 16 to concisely define a custom monadic context based on standard monad transformers.
- *No implicit prelude* means we will not be making use of the standard **Prelude** module from the **base** package.
- *Quasi-quotes* enables an alternate form of string literal syntax that we will use for writing ASCII strings beginning in chapter 5.

The **ghc-options** field specifies what GHC flags we use:

- **-Wall** enables all warnings, which we always do because this includes some particularly important ones such as detecting when a **case** expression is missing some cases.
- **-fdefer-typed-holes** allows us to write an underscore _ in place of an expression, which we will use to write definitions that aren't quite finished yet. Such an underscore is called a "hole."

We list what libraries we need in the **build-depends** field. The only library we'll be using is **sockets-and-pipes**, which re-exports all of the modules used in the book. If you go off exploring on your own and want to use other libraries that we have not included in the **sockets-and-pipes** package, you can add additional entries to this **build-depends** list, separated by commas.

library/Book.hs Start this file as follows:

```
module Book where

import Relude
import qualified System.Directory as Dir
import System.FilePath ((</>))

getDataDir :: IO FilePath
getDataDir = do
    dir <- Dir.getXdgDirectory Dir.XdgData "sockets-and-pipes"
    Dir.createDirectoryIfMissing True dir
    return dir
```

This is where you will enter all of the code and exercise solutions. All of the definitions given in the book are named uniquely, so you should not need to remove anything as you go along; keep everything for future reference.

The first import is **Relude**, an alternative prelude module which comes from the **relude** package. Relude is similar to the standard prelude, but it includes a few extra conveniences and will spare us some additional imports.

There are **import** declarations interspersed throughout the book; each time you encounter one, add it to the import list at the top of **Book.hs**. When we use an identifier that is re-exported by **Relude**, we do not give an import declaration since it would be unnecessary, but we may mention what module it originally comes from.

Once you have **book.cabal** and **library/Book.hs**, you can then use the command **cabal repl** to load your code into GHCi. We will use the REPL to query for type information and to run example programs. Try it now:

```
λ> getDataDir
"/home/chris/.local/share/sockets-and-pipes"
```

The file path you see here will vary based on your system. Make note of it; this is where you will be storing data files that we use as examples throughout the book.

Proximity to reality There is always some tension between the code one writes for learning and the code one writes for some other purpose.

- Our recommendation of putting all the code into a single module is a departure from normal practice. Organization is a critical aspect of manageably developing software, but a single file keeps best with the linear progression of a book, and so we set aside our usual emphasis on designing small, cohesive modules.
- The code that we give in this book is not annotated with any comments; the surrounding context of the book serves as the explanation that comments in code would normally provide. You are encouraged to insert your own comments in your **Book.hs** file, for the sake of note-taking and practice in writing API documentation.
- Our code is wrapped into short lines, and some local variables have highly abbreviated names; these superficial choices follow from the constraints of print and are not necessarily intended as a recommended style in general.

In all other matters, the code in this book is intended as a presentation of good Haskell as written in practice. The words of an old theater teacher echo in my mind – "What you do in rehearsal is what you'll do in the performance." – and so we have aimed here to avoid oversimplification and to make the same choices one might make on the job.

Chapter 1

Handles

Let us begin with some perspective on a process's position in the world.

1.1 The necessity of indirection

A typical running computer contains multitudes. Some processes correspond to things that you see on the screen; others work quietly behind the scenes. Your text editor, your terminal, each tab in your web browser, the synchronization of your system clock with other machines, that icon in the corner that shows you the signal strength of the wifi – each of these is controlled by a separate process. For the purposes of this book, we'll oversimplify a little and say that a software ***process*** is an instance of a program that is running, although the distinction is not always so simple or clear – for example, a single instance of a web browser may create a separate process for each tab.

You can start as many processes as you want (within reason). But while this multiplicity exists in software, the hardware is fixed. A machine might have only one screen displaying graphics, one speaker playing sound, one chip storing all of the files, one cable or wireless transmitter linking the machine to the internet. It seems surprising, then, that computers can work at all – as if a hundred chefs may simultaneously cook a hundred soups sharing only a single pot among them, or as if a hundred orators may be heard delivering a hundred speeches at once at a single podium. Allowing many tenants to share the same facilities without interfering with one another is called ***multiplexing***. This is the job of an operating system: to coordinate shared use of physical resources, weaving together the actions performed by many separate processes.

When we say that a program performs an action, we are usually talking about interacting with physical resources. Reading and writing files from the hard drive, downloading from

and uploading to the internet, listening to and playing sound – inputs and outputs – *I/O*: all of these actions are mediated by the operating system. To say that a program *does* something ascribes more agency to it than it really has; the only thing its process can ever really do is issue requests for the operating system to do things on its behalf. These requests are called *system calls*.

Each operating system has its own set of system calls that programs running on it have to use to do I/O. You can run **man 2 syscalls** at the command prompt on a Linux machine to see the complete list of system calls supported by the Linux kernel. We will not give much attention to the differences between the operating systems because the Haskell libraries we use take care of these differences for us; the **base** and **network** libraries will automatically use whatever system calls are appropriate for the specific platform our programs are compiled for.

1.2 Writing to a file

If we consider I/O actions as lines of dialogue in a conversation between process and operating system, then we might think of a *handle* as an identifier for the conversation.

A handle for a file is referred to by Microsoft Windows as a *file handle* and by Linux as a *file descriptor*, often abbreviated as **fd**. In the Haskell libraries we use, you will see both terms used interchangeably. We prefer 'handle,' but both are misleading. In a strained attempt to justify the metaphor: a handle on a resource is a temporary means of grabbing onto it, like the handhold provided by an ice climber's axe.

Our first demonstration involving a file handle is a brief **IO ()** action that uses the basic operations of the **base** library to write two lines of text to a file:

```
import qualified System.IO as IO
```

```
writeGreetingFile = do
    dir <- getDataDir
    h <- IO.openFile (dir </> "greeting.txt") WriteMode
    IO.hPutStrLn h "hello"
    IO.hPutStrLn h "world"
    IO.hClose h
```

Enter this into your code file. Open GHCi and run **writeGreetingFile**, then look at the file that it created. (Remember that you can run **getDataDir** to see the path where files are stored.)

Getting a handle The first argument to the **openFile** function is the path of the file we want to write. The second argument, whose type is **IOMode**, is where we declare what we intend to do with the file: read its contents, write new contents, or both. **IOMode** is defined in **System.IO** as:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Why do we have to specify upfront whether we're opening this file for reading or for writing? Remember that we aren't opening a file directly; we're asking the operating system to open a file for us. The answer lies in the OS's mediation and multiplexing responsibilities:

- The file system may have security restrictions that, for example, permit our process to read this file but not to write to it. The OS is responsible for enforcing this sort of access policy, and it does this permission check at the time a file is opened.
- Ours might not be the only process that accesses this file. Two processes simultaneously reading a file is fine, but two processes attempting to write to the same file at once could be trouble. The OS keeps track of all processes' file handles, and whether each is for reading or for writing, to prevent any conflicts.

The result of **openFile** will be a handle, which we've called **h** in this example. **hPutStrLn** has to know the handle in order to know where to put its strings.

The Haskell **openFile** function corresponds to the **open** system call in Linux. You can run **man 2 open** at the command prompt on a Linux machine to see the documentation for this system call. The return type of **open** is **int**, a number that the OS has assigned to you, like when you file an insurance claim and your insurance company gives you a claim number. *Please provide this number in all further communications regarding this claim*, they will tell you. That's what the file handle is. We have to provide this number as an argument to all subsequent system calls that pertain to this particular OS-mediated interaction with this file. The Haskell **openFile** function wraps up this number into a value of type **Handle**, which is a bit more complicated but serves the same role as a conversation identifier.

Writing to a handle If you've seen a Haskell program, you've probably seen the **putStrLn** function in the **Prelude** module. We can use it like this:

```
helloWorld = IO.putStrLn "hello world!"
```

What you may not have known is that this program is using a handle. **putStrLn** is a specialization of a more general function called **hPutStrLn** in the **System.IO** module. (The 'h' stands for 'handle.')

```
IO.hPutStrLn :: Handle -> String -> IO ()
```

putStrLn is defined in terms of **hPutStrLn** and **stdout**, a handle we discuss shortly below.

```
IO.putStrLn :: String -> IO ()
IO.putStrLn s = hPutStrLn stdout s
```

Even in the simplest *hello world* program, handles are there, just under the surface. The small **helloWorld** program above can be expressed equivalently as:

```
helloHandle = IO.hPutStrLn IO.stdout "hello world!"
```

The **Handle** parameter lets us write strings to destinations other than the standard output stream **stdout**. We saw an example of that earlier when we wrote to a file called **greeting.txt**.

What is stdout? When the OS starts a process, it creates by default a few "standard" places for the process to read and write. The *standard output stream* is one of those, and **stdout** is the handle for it.

```
IO.stdout :: Handle
```

Each process has its own **stdout**. What happens when the process writes to its standard output stream? It depends on context. We often think of it as "how you print messages to the terminal," because if we run a program at a command prompt, that's what will happen.

Suppose you were to place the following into a file named **hello-world.hs**:

```
module Main (main) where

import qualified System.IO as IO

main = IO.putStrLn "hello world!"
```

When you run it, you see that it prints output to the terminal.

```
$ runhaskell hello-world.hs
hello world!
```

But remember, a process never does anything directly! All I/O goes through the OS, and **stdout** is no exception. If we start the process in a context where its output is piped to a file, for example, then what it writes to **stdout** doesn't display in the terminal. That same **putStrLn** action ends up writing to a file instead.

```
$ runhaskell hello-world.hs > greet.txt
```

```
$ cat greet.txt
hello world!
```

Background processes like servers often write their log output to **stdout** with the expectation that the OS will store all of the daemons' logs.

Closing a handle Once we're done writing, we use **hClose** to tell the OS that we're done with the file handle. This isn't really necessary in our little demo program, because when the process ends, the OS will close all of its handles automatically anyway. But for long-running processes that may end up doing a lot of file operations, it can be important to make sure that handles get closed. The OS has to use memory to keep track of all of these handles, and if the number of handles associated with your process just keeps rising because you're not closing them, eventually your operating system will become enraged and refuse to keep giving you more.

It is okay to run **hClose** on a handle more than once. After the first time, it has no effect.

1.3 Exceptions

Even in a program so simple as this, we can concern ourselves with the possibility of exceptions. What, you might ask, could possibly go wrong in a program as simple as ***hello world***? The **openFile** action can fail if filesystem permissions do not allow us to write to the file path. The **putStrLn** action can fail if the filesystem is backed by a storage medium that is full.

Try altering the file-opening line of **writeGreetingFile** as follows. This causes the operation to fail, because the system cannot create the **greeting.txt** file at this path because the parent directory does not exist.

```
    h <- IO.openFile (dir </> "nonExistentDirectory/greeting.txt") WriteMode
```

```
λ> writeGreetingFile
*** Exception:
/home/chris/.local/share/sockets-and-pipes/nonExistentDirectory/greeting.txt:
openFile: does not exist (No such file or directory)
```

tryAny The default behavior when an exception occurs is to halt the action and print the sort of error message that you see above. Whenever we want to change that behavior, we will use **tryAny** from the **safe-exceptions** package:

```
import Control.Exception.Safe (tryAny)
```

```
tryAny :: IO a -> IO (Either SomeException a)
```

To "try" an action is to bring its demons out from the shadows into a place where we must confront them. The type of **IO Handle** indicates that an action returns a **Handle**, but it says nothing about the fact that you might get an exception instead. **IO (Either SomeException Handle)** doesn't let us get to the **Handle** until we reckon with what our software should do with an exception.

One simple reason we might want to customize exception-handling behavior is to make the output of a command-line application more uniform. Perhaps you want all messages indicating problems to be printed in bold red. For a fully graphical application, exception handling is essential, since the user cannot see the output printed by the default behavior at all. For an easier example, suppose your program leans heavily into emoticons:

```
writeGreetingTry :: IO ()
writeGreetingTry = do
    dir <- getDataDir
    IO.hPutStrLn IO.stderr "About to open the file :/"
    openResult <- tryAny $ IO.openFile (dir </> "greeting.txt") WriteMode
    case openResult of
        Left _ -> IO.hPutStrLn IO.stderr "Cannot open file to write :("
        Right h -> do
            IO.hPutStrLn h "hello"
            IO.hPutStrLn h "world"
            IO.hClose h
            IO.hPutStrLn IO.stderr "Done :)"
```

One should always take care to ensure that problems are *visible* in some way. For command-line applications and fooling around in GHCi, the default exception behavior which prints the exception's error message is decent in this regard. When you start catching exceptions with **tryAny**, you become responsible for replacing the default behavior with something of your own devising! Make sure you don't write software in which important error information is left invisible, leaving users no recourse to understand what went wrong when the unexpected happens.

Asynchronous exceptions There is one class of exception that **tryAny** does not catch. Even when not engaged in any risky behavior, an action can *receive* an exception at any time, when someone (or something) decides to kill it. This happens, for example, when you send an interrupt from a terminal program (typically using the "ctrl+C" keyboard shortcut), or when one thread in the program throws an exception to another (see the **throwTo** function in the **Control.Exception** module). An exception received from without is called an *asynchronous* exception. Any **IO** action's chief responsibility when receiving an asynchronous exception is to halt quickly, because this is necessary if we are to have processes that are responsive to kill signals. It is terribly frustrating when you try to stop an application and it just won't quit! Catching asynchronous exceptions is a risky proposition that should be left only to dedicated libraries that really know what they're doing.

1.4 Diligent cleanup

We want to make sure our handles get closed when we're done with them, no matter what happens. In this light, there is a subtle problem with **writeGreetingFile**. Although it does close the file handle, this is only assured if nothing goes wrong. If an exception is raised during the printing of the "hello world" text, the **writeGreetingFile** action terminates without closing the handle. In a single-threaded program, this is not a big deal; when the exception is thrown, the process halts, and the operating system cleans up any open handles. However, a Haskell program that employs concurrency, such as the server we will be writing, has many **IO** actions running at once, and the process can stay alive even if an exception halts one of its threads.

To be sure that we deal with handle-closing reliably, it might seem like a good idea to use **tryAny** to wrap up the steps between **openFile** and **hClose** to ensure that no exception can be thrown between the moment when the handle is opened and the moment that it is closed. However, this approach is still not a full guarantee because **tryAny** does not catch asynchronous exceptions.

```
writeGreetingSafeAttempt :: IO ()
writeGreetingSafeAttempt = do
    dir <- getDataDir
    h <- IO.openFile (dir </> "greeting.txt") WriteMode
    _ <- tryAny do
        IO.hPutStrLn h "hello"
        IO.hPutStrLn h "world"
    IO.hClose h
```

We will instead adopt **ResourceT**, a more reliable and convenient device provided by the **resourcet** library. Add the following imports. We will discuss each new bit of the new greeting program in detail below.

```
import Control.Monad.Trans.Resource (ReleaseKey, ResourceT, allocate, runResourceT)
```

```
writeGreetingSafe = runResourceT @IO do
    dir <- liftIO getDataDir
    (_releaseKey, h) <-
        allocate (IO.openFile (dir </> "greeting.txt") WriteMode) IO.hClose
    liftIO (IO.hPutStrLn h "hello")
    liftIO (IO.hPutStrLn h "world")
```

ResourceT The type of the **do** expression above is **ResourceT IO ()**. The T in **ResourceT** stands for "transformer" because **ResourceT** is a *monad transformer* – if **m** is a monad, then **ResourceT m** is a monad as well. **ResourceT IO** represents the concept of **IO** that has been transformed or modified by adding a certain safety feature. A **ResourceT IO** action is much like an **IO** action, but it is augmented with a register of resources that are guaranteed to be closed when the **ResourceT IO** action concludes.

Above we have used three new functions: **allocate**, **liftIO**, and **runResourceT**. Before we talk about what each means, we must first reckon with the many constraints in their type signatures.

```
liftIO       :: MonadIO m       => IO a -> m a
allocate     :: MonadResource m => IO a -> (a -> IO ()) -> m (ReleaseKey, a)
runResourceT :: MonadUnliftIO m => ResourceT m a -> m a
```

We said a moment ago that a **ResourceT IO** action is much like an **IO** action. Whenever one type is like another, there are probably typeclasses involved. In the constraints above, we

see three classes: **MonadResource**, **MonadUnliftIO**, and **MonadIO**. Wherever there are monad transformers, we find this sort of abundance of polymorphism, because it is often (but not always) the case that if some operation can be performed in the base context, then it may also be performed in the transformed context.

MonadIO This class describes any monad that an **IO** action can be lifted into. Its sole method is:

```
liftIO :: MonadIO m => IO a -> m a
```

ResourceT IO belongs to the **MonadIO** class, which tells us that **ResourceT IO** is in some sense more powerful than plain old **IO**; anything that you can do in **IO**, you can also do in **ResourceT IO** by *lifting* the ordinary **IO** into a resource-safety-augmented **ResourceT IO** context.

IO also belongs to the **MonadIO** class, which expresses the comically trivial fact that an **IO** action is an **IO** action.

```
instance MonadIO IO where
    liftIO x = x
```

Such an instance may usually be found whenever a class's role is only to express that values of one type may be converted to another type. Although silly-looking, an instance like this does serve a purpose. When we encounter a function with a **MonadIO** constraint, it means we can use that function in any context that **IO** can be lifted into. But in situations where we do not need any augmentations, it also means that we can use that function with plain old unadorned **IO**, thanks to the humble **MonadIO IO** instance.

liftIO originally comes from **Control.Monad.IO.Class** and is re-exported by **Relude**.

MonadUnliftIO We do not need to understand this class. Let it suffice to say that **IO** has an instance for it. You don't necessarily need to know what every constraint in a polymorphic type signature means. Sometimes all you need to do is look at the class's instance list to verify that the concrete type you're interested in using satisfies the constraint. This can be an important skill when reading Haskell API documentation.

MonadResource This class describes the special safety augmentations that the **resourcet** library provides. **ResourceT IO** belongs to this class, but regular old non-augmented **IO** does not. The **MonadResource** class exists because **ResourceT IO** can have further monad transformers applied to it, and the transformed monad in many cases also supports the polymorphic **allocate** function.

It is easy to get lost in a sea of typeclasses. If you begin to feel overloaded, focus on concretized type signatures. The three polymorphic functions listed earlier, as we will use them in our hello-world program, are specialized as follows:

```
liftIO       :: IO a -> ResourceT IO a
allocate     :: IO a -> (a -> IO ()) -> ResourceT (ReleaseKey, a)
runResourceT :: ResourceT IO a -> IO a
```

Whenever you learn about a new type, pay attention to which functions introduce that type and which functions eliminate it. For example, with **ResourceT**:

- Introduction: **liftIO** and **allocate** create **ResourceT** values.
- Elimination: **runResourceT** takes an **ResourceT** as its argument and turns it into something else.

This tells us what the general structure of a program using the type will look like. To use **ResourceT**, we construct a **ResourceT** value using **liftIO** and **allocate**, and then we turn it into **IO** using **runResourceT**.

allocate The **allocate** function has two parameters:

1. `IO a` – An action that creates/opens/acquires a resource;
2. `a -> IO ()` – An action that destroys/closes/releases the resource.

When a **ResourceT** action performs an **allocate**, it runs the "open" action immediately and holds onto the "close" action for later. At the end of the **ResourceT** computation, it runs all of the "close" actions that have been registered. The closing phase is guaranteed to take place even if an exception interrupts the process.

release In circumstances where it is known that the **Handle** is no longer needed *before* the end of the **ResourceT** computation, you can use the **release** function to close it early. We give it the **ReleaseKey** that we obtained from the **allocate** function.

```
release :: MonadIO m => ReleaseKey -> m ()
```

Behind the scenes, the **ResourceT** context is maintaining a register of every cleanup action that needs to run once the action is over. If you are done using the file handle before the end of the action, it is tidier to use **release** instead of applying **IO.hClose** directly, because the **release** also scratches off the relevant cleanup action from the list.

Our small program does not need to make use of **release**, so we simply discard the **ReleaseKey** that **allocate** provides.

runResourceT One never constructs a **ResourceT** computation for its own sake; the ultimate purpose is to "run" the computation, which is a cute way to describe converting it to an **IO** action that can be run in GHCi or used as **main** to compile an executable.

We prefer to use **runResourceT** with a type application, writing **runResourceT @IO** to specify **IO** as the "base" monad. This type application is what allows the compiler to infer the type signature of **writeGreetingSafe** that we have not given explicitly:

```
writeGreetingSafe :: IO ()
```

Throughout the rest of this book, whenever we open a file handle or other resource that needs to be closed, the **runResourceT** function will be involved.

1.5 MonadIO

We have discussed **MonadIO** class and what **liftIO** means. We add here a few pragmatic notes about a programmer's relationship to it.

When introducing a new monad transformer like **ResourceT** into an **IO** sequence, every line under the **do** keyword whose type is not already of type **ResourceT IO** will have to be lifted into **ResourceT IO** using **liftIO**. (We saw this in the **writeGreetingSafe** program earlier.) This is a function that we often forget to apply, and one eventually learns to quickly recognize the type error "Couldn't match type **IO** with **ResourceT IO**." An error message like this often indicates that a missing **liftIO** must be inserted.

A design question arises whenever a definition has an **IO** type, such as the **helloWorld** action.

```
helloWorld :: IO ()
helloWorld = IO.putStrLn "hello world!"
```

Such a definition can always be made polymorphic:

```
helloWorld :: MonadIO m => m ()
helloWorld = liftIO (IO.putStrLn "hello world!")
```

In the wild jungle of Haskell libraries, we find disagreement about whether this should always be done as a matter of course. For example, the **System.IO** and **Prelude** modules make no use of the **MonadIO** class whatsoever, whereas **Relude** provides lifted **MonadIO** variants of many of the same functions.

Neither choice is obviously preferable in all circumstances. Polymorphic **MonadIO** type signatures are more convenient for users of monad transformers like **ResourceT**, since they remove the need to sprinkle **liftIO** throughout the code. Monomorphic **IO** types are sometimes easier on users who do not need lifted **IO** because polymorphism can weaken type inference, requiring type annotations or explicit type applications, such as **runResourceT @IO** as we have written here. The polymorphic choice may also induce more complicated error messages if type errors arise.

1.6 Exercises

Exercise 1 – File resource function Define a function with the following type signature:

```
fileResource :: FilePath -> IOMode -> ResourceT IO (ReleaseKey, Handle)
```

Rewrite **writeGreetingSafe**, this time using the **fileResource** function we defined instead of using **allocate** directly.

Keep this **fileResource** function around, because we will continue to make use of it in later chapters.

Exercise 2 – Showing handles **Handle** has an instance of the **Show** typeclass, which means you can use this function:

```
show :: Handle -> String
```

However, **show** doesn't give you much of the information you might want to see when you look at a handle. This is because a **Handle** is a messy real-world mutable sort of entity that requires I/O to ascertain its current state, and **show** is a puny pure function that does not have **IO** in its type and cannot look up information about a mutable object. So in addition to the **Show** instance, we also have the **hShow** function:

```
IO.hShow :: Handle -> IO String
```

Write a program that uses **show** and **hShow** on a few handles to experiment and see the difference between these two functions' outputs.

```
handlePrintTest :: IO ()
handlePrintTest = _
```

Exercise 3 – Exhaustion How many file handles can you have open at once? Fill in the two holes below and run **howManyHandles** to find out.

```
howManyHandles = runResourceT @IO do
    hs <- openManyHandles
    putStrLn ("Opened " <> show (length hs) <> " handles")
```

```
openManyHandles :: ResourceT IO [Handle]
openManyHandles = _
```

```
fileResourceMaybe :: ResourceT IO (Maybe Handle)
fileResourceMaybe = do
    dir <- liftIO getDataDir
    result <- tryAny do
        _
    case result of
        Right x -> return x
        Left e -> do
            print (displayException e)
            return Nothing
```

The **openManyHandles** action should keep reopening a file until **IO.openFile** throws an exception, and it should return a list of all the handles that were opened. Recursion will be useful for making this action repeat in a loop.

This action will need to *catch* the exception that **openFile** will throw when the maximum number of file handles has been reached. Use the **tryAny** function. The type signature we gave for **tryAny** earlier was actually a simplification; it is polymorphic. For this exercise, **tryAny** specializes to:

```
tryAny :: ResourceT IO (Maybe Handle)
    -> ResourceT IO (Either SomeException (Maybe Handle))
```