Extracted from:

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming

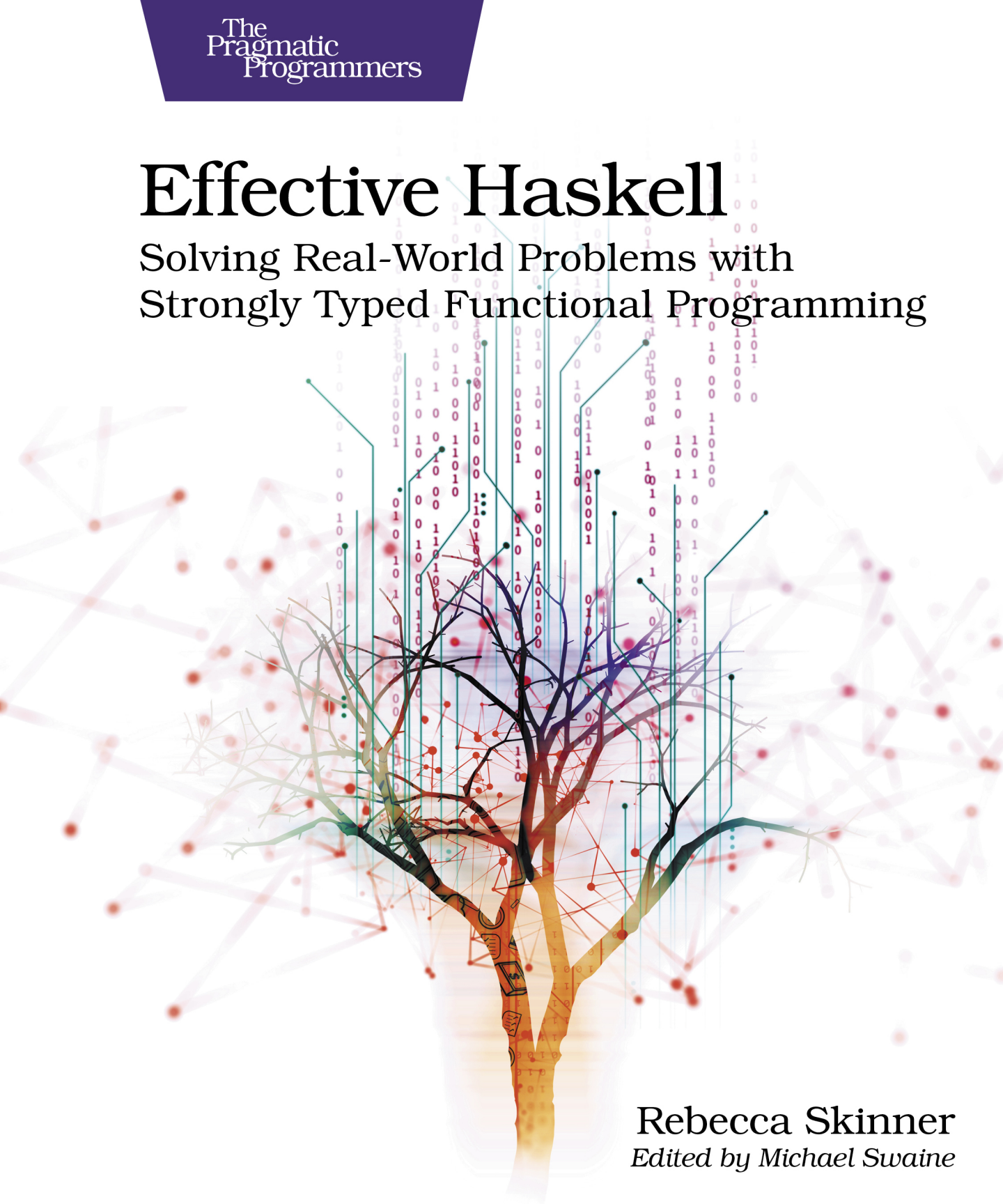
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming



Rebecca Skinner

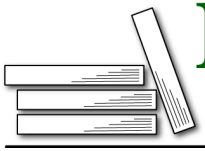
Edited by Michael Swaine

Effective Haskell

Solving Real-World Problems with Strongly Typed Functional Programming

Rebecca Skinner

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-934-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2023

Understanding Space Leaks

Space leaks are a type of error we can run into in Haskell when laziness starts to work against us, and we start to see a large number of thunks accumulated that are not evaluated. This can manifest in several different ways, including causing our programs to use too much memory, to have poor or unpredictable performance patterns, or more rarely, to crash at runtime with a stack overflow. What we've seen so far when trying to run our directory traversal program is an example of how space leaks can make themselves known when the performance isn't what we expect. Specifically, we would expect generating the character histogram to take a lot of time, but in fact it takes almost no time at all.

The idea that a space leak can show up as a problem with unexpected performance characteristics can be counterintuitive, but as we look into what happened with our program in this case it will start to be clear just why the unusual performance was an indicator of a space leak.

Before we dive into the code and start working on a fix, let's get some hard data. When we suspect that we have a space leak, it can be helpful to look at information about the amount of memory we're allocating, and what the garbage collector is doing. Even if the data doesn't initially tell us where to look for the error, it gives us a baseline to measure against, so we can see if the changes we're making are actually having a positive impact on the runtime characteristics of the application.

We can't get the kind of information that we need from ghci, so let's create a new file and make sure that we have main defined so that we can compile the application as a stand-alone program:

```
import System.Environment (getArgs)

main :: IO ()
main = getArgs >>= directorySummaryWithMetrics . head
```

If you haven't already created a new cabal project for this program, you can take a minute to do so now, or you can compile the program directly with ghc. In either case, make sure that you've enabled -O2 level optimizations. GHC is able to do a number of optimizations, and we want to avoid spending too much time chasing down optimizations that the compile will take care of for us anyway.

```
$ ghc -O2 DirectorySummary.hs -o DirectorySummary
```

Once you've built the program, we want to run it, but instead of running the program like normal, we're going to pass in some extra flags to the Haskell runtime so that we can ask it to collect some information about memory usage as our program is running. Flags like this that we use to control the way the Haskell runtime works, or to ask it for some extra information about our program, are called *RTS flags*. RTS flags are normal command line flags, but we need to differentiate between options we want to pass to the Haskell runtime and the options that we want to pass to our program. To do so, we start by passing in the special +RTS argument. This argument will cause the runtime to interpret all the arguments that it sees as arguments to the runtime system, until it sees the -RTS argument. This lets us pass in as many arguments as we want to the runtime system without our application having to know about or handle them.

In our particular case, we only want to pass a single RTS flag, -s. This flag will ask the runtime to generate summary statistics about the memory utilization of our application:

```
$ ./DirectorySummary +RTS -s -RTS ./example-dir/
```

When you run the program with this RTS flag you'll get all of the normal output you'd expect, and then at the end before your program exits you'll see some output like this:

```
 1,911,494,744 bytes allocated in the heap
    44,005,608 bytes copied during GC
    12,816,336 bytes maximum residency (10 sample(s))
     6,051,224 bytes maximum slop
            39 MiB total memory in use (3 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0   1764 colls,  0 par    0.026s   0.026s     0.0000s    0.0004s
Gen  1     10 colls,  0 par    0.007s   0.007s     0.0007s    0.0012s

INIT    time    0.000s  (  0.000s elapsed)
MUT     time    0.363s  (  0.363s elapsed)
GC      time    0.033s  (  0.033s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time    0.396s  (  0.396s elapsed)

%GC     time       0.0%  (0.0% elapsed)

Alloc rate    5,262,713,737 bytes per MUT second

Productivity  91.7% of total user, 91.6% of total elapsed
```

There's a lot of information here that we won't cover in this book, but you can refer to the GHC User Guide[1] for comprehensive documentation on the meaning of all these fields. For the moment we're going to focus on the *maximum residency* field, which tells us the amount of memory that our program was actually using at its peak.

If you look at the total size, in bytes, of all of the data in your example directory, you'll notice that it is fairly similar to the total residency of our application. For example, if we look at the total number of bytes in all of the files in example-dir, we'll see that they total about 12.3 megabytes, which is a little bit less than the total residency of our application, but suspiciously close:

```
$ du -s -B 1 ./example-dir/
12365824        ./example-dir/
```

The fact that our maximum residency is so similar to the size of all of the files in our directory can start to give us a hint about what has happened. When we read the contents of a file in so that we can calculate the character histogram, we're not freeing that data right away. Instead, we're keeping all of the files in memory. The fact that we see observable delay before the histogram is printed out on the screen gives us a bit more information: we're reading all of the files, but not actually calculating the histogram until we're ready to print it out. It seems like, in this case, laziness might be causing trouble. Let's take a look at what's going on, and in the next section, we'll look at a few ways to address this particular type of problem.

1. https://downloads.haskell.org/ghc/latest/docs/html/users_guide/index.html

Laziness, Strictness, and IO

The root cause of the problem we've run into is that we're mixing lazy and strict values, and it's causing our program to act unexpectedly. Let's take a look at our histogram calculation code again for reference:

```
traverseDirectory metrics root $ \file -> do
  contents <- timeFunction metrics "TextIO.readFile" $
                TextIO.readFile file

  -- Omitting some things here

  timeFunction metrics "histogram" $ do
    oldHistogram <- readIORef histogramRef
    let
      addCharToHistogram histogram letter =
        Map.insertWith (+) letter 1 histogram
      newHistogram = Text.foldl' addCharToHistogram oldHistogram contents
    writeIORef histogramRef newHistogram
```

The first thing that we need to keep in mind here is that TextIO.readFile is a strict function. Whenever we call it, we're going to get the entire contents of the file brought into memory. Similarly, combining IO actions using (>>=) or in a do block is always strict. As we're traversing the directories, we're always going to read the contents of the file before we write an update to histogramRef or before we move on and read the contents of the next file.

The second thing that we have to keep in mind is that Haskell is lazy by default, so all of the things that don't have to be strict are going to generate *thunks* instead of strictly evaluated values. That means that whenever we create a new histogram, we're not really computing the value of a brand new histogram, we're just creating a new thunk that can compute a histogram when a histogram is needed:

```
newHistogram = Text.foldl' addCharToHistogram oldHistogram contents
```

Perhaps unintuitively, a value to an IORef is not strict. When we call writeIORef, we're not forcing the value newHistogram to be computed, instead we just write the thunk into the reference.

The last thing to keep in mind is a thunk keeps around references to everything that is needed to compute a value. In this case, each thunk we're writing into the reference is keeping a reference to the previous thunk that was stored in the IORef and a reference to the contents of the text file we've just read. Since we have a reference to the contents of the text file, that data can't be garbage collected. Since we have a reference to the previous thunk, which in turn has a reference to the contents of its text file, that text file can't be

garbage collected either. By the time we've finished traversing the directory, we have a chain of thunks that are keeping open references to all the data from all the files we've opened—plus a bit of overhead for the other calculations we need to do.

The solution to this problem is to reduce the amount of laziness in our program. If we can compute the value of the histogram thunk immediately, then we will no longer need to keep references to the contents of the file or the previous thunk, and the garbage collector can clean everything up for us. This is a common enough problem in Haskell programs that we have not just one, but several different approaches we can use to solve the problem. Before we dive into reviewing the options though, let's take a slight detour to understand exactly what we mean when we're talking about strictness, laziness, and what it means to evaluate an expression. This will give us the tools to better understand when and how to introduce strictness, and also make sure we're better prepared to avoid this type of space leak in the future.