LEARNING HASKELL WITH NETWORK PROGRAMMING

CONTENTS

FOREWORD

Haskell, as a language, has a bit of a loaded reputation.

A lot of programmers assume that Haskell is that "mathey language", and once they hear mention of functors, monoids, monads, and arrows, they end up with a conclusion that it stands as a a tool that might have some interesting ideas, but is too academic to be suitable for any kind of practical use.

This is an easy viewpoint to have. There is a steep, non-trivial learning curve to the minutia of functional programming, and Haskell is among the most dogmatic of the functional languages; to understand Haskell is to understand the basics of algebra, Category Theory, Type Theory, and Set Theory. Consequently, many programmers assume that people learn Haskell so that they can look smart.

While this is certainly true (and is why I picked up the language initially), functional programming has started to become fairly popular in the last couple years, and many people have started to appreciate the benefits of concepts like immutability, a clear type system, and the mathematical safety that programming in a functional style provides. The rising popularity of languages like Scala, Rust, and F#, as well as the newer functional features being added to Java 8, C++, and C# has shown us that there is a clear market for functional programming in the field of software engineering.

Learning Haskell is about more than just the language. Haskell has become the unofficial "testing ground" for the cutting edge features of computer science.

New methods of parallelism, thread safety, parsing, and nearly every other new programming theory often come over to Haskell first.

But what you get with Haskell isn't just access to functional features. Indeed, if you wanted an excuse to play with functional programming, the barrier of entry to do so is much lower with JavaScript.

What you get from Haskell is a purely devoted, unambiguous entry to a functional way of structuring your programs, with all the benefits and annoyances that come with that. You get all the frustration of having to avoid state, but also all the benefits of guarantees that your functions will work exactly the same after being run a trillion times. You get all the frustration of having to use recursion instead of loops, but all the benefit of knowing that you'll never be forced to deal with a `null` reference error.

The main feature of Haskell is something you do not get with many other languages: guarantees. Mathematics is one of the very few things in life that boasts in its proofs and correctness and by taking a purely mathematical approach allows you to reap all the benefits of all that scary topics that I mentioned before. And with those guarantees, you find that Haskell is a language *especially* well-suited for production systems that need to *keep working*.

Upon learning Haskell, it's not uncommon for a person to re-evaluate and change their entire coding process to use a more functional style, or to start utilizing the functional features of their favorite language.

Being forced to view things from a different, strange angle can be a godsend in how you approach problems; sometimes approaching certain problems from a functional perspective can greatly simplify the problem. Recursion is a useful, often-underutilized tool that becomes easier to understand once confronted with a world without loops. Optional types give compile-time checks to that completely infamous `null` reference exception. Higher-order functions give a level or modularity that is nearly impossible to replicate with more "traditional" imperative coding styles.

There are plenty of reasons to learn functional style, and many of them will be described in this book, but if you are to take anything out of this book, it should be that there is value in the extravagant and theoretical. I hope you enjoy it, and find that something like Haskell can open the door to the other interesting new paradigms in this whacky field we call "Computer Science".

CHAPTER 1

GETTING EVERYTHING INSTALLED

It can be incredibly difficult to know where to even start when doing Haskell. The documentation on the Haskell Wiki, while very good, is designed for a more academic audience than most of the more mainstream langauges. Challenges tend to arise trying to figure out *which* compiler to install, let alone how to go about structuring things or installing libraries.

This is probably why often the biggest hurdle to cross when getting started with a new programming language is simply getting the compiler, platform tools, and editor set up for beginning your development, if for no other reason than the fact that the territory is still entirely new and it can be difficult to know when you did something incorrectly.

Until very recently (at least as of this writing), Haskell has been an especially difficult platform to work because of its ambiguous ways of setting everything up, especially on Windows. Different versions of different point releases of Haskell compilers crowd their way through the package managers, making code more challenging to port between systems than it had any right to be.

Before getting too far into how to go about installing things, we should define some key terms that will be used frequently throughout the book.

1.0.1 GHC

The Glorious Haskell Compiler, or GHC for short, is the *de facto* standard compiler for doing Haskell work. It has been in development since the late eighties, and is usually what people mean when they say "Haskell". It survives by the contributions of dedicated enthusiasts and from funding from Microsoft Research, and is generally the first Haskell compiler to get new features (although they tend to be hidden away in compiler extensions).

1.0.1.1 Some Notes About GHC There is an old saying about Haskell by Simon Peyton Jones: "Avoid Success At All Cost". The meaning of this statement was to imply that Haskell should remain free as a platform for new ideas and doing things the "right" way, and not cave to the pressures of industry demands.

This is all well and good, but it also means that the developers of GHC can and will often introduce new features and rules for the language as often as they want, which can lead to breaking changes between differing versions of GHC, sometimes even for point-releases.

Several tools, such as Stack, have done a lot of great work to minimize the damage that these changes can cause, and it should be noted that the later versions of GHC have been doing much better at avoiding huge breaking changes, but it is something you have to be aware of.

1.0.2 Cabal and Stack

While GHC does a good job at compiling your Haskell programs, it doesn't dictate how to structure the project or handle library dependencies. GHC is exactly what its name states: a compiler.

Like many modern languages, most Haskell developers eschew the use of Makefiles and build scripts, instead utilizing a folder structure and config file.

Cabal was the first major system to allow such a style for Haskell, and remains a popular tool today for such reasons. Cabal gives you a structure for building your projects, gives you simple commands like `cabal build` and `cabal repl`, giving you a simple-to-read project file for handling dependencies from Hackage (the standard repository for Haskell libraries), basic sandboxing, compiler extensions, and build parameters.

However, Cabal does nothing to address the fragmentation of GHC that was mentioned earlier. Cabal's installation is predicated on the notion that you already have a functioning GHC installed, and even installing Cabal itself can lead to some incompatibility between versions. This, compounded with the almost complete lack of management of transitive dependencies has lead to something people in the Haskell trade have labeled "Cabal Hell", and has been the bane of nearly every veteran the language has. It was not and still is not difficult to find people on Hacker News and Reddit complain about how they gave up on Haskell due to problems with getting Cabal to work.

In 2015, the people at FP Complete addressed this problem with their new build platform: Stack. Stack still used Cabal for things like dependency management behind the scenees, but gave Haskell a feature it had never had before: reproducible builds.

Stack allows the user declare which version of GHC they want to use for their project, on a per-project basis, and even allows them to declare it in the "shebang"

at the beginning of scripts. Like Cabal, it gives a structure to your project, gives you access to commands like `stack build` and `stack ghci`, which will generally "just work" on your machine.

They also built an alternative repository called "Stackage", which, while similar to Hackage, audits packages to greatly reduce the likelihood of a transitive dependency conflicts, almost (but not completely) eliminating the potential for Cabal Hell.

1.0.2.1 A note about operating-system package managers I should also make this point abundantly and overwhelmingly clear: do *not* install GHC through your operating system's package manager like `apt`, `pacman` or `brew`. The versions of GHC included in these repositories tend to be incredibly out of date, and are installed system-wide, thus making it difficult to install differing versions of GHC or Cabal for different project.

It is best to ignore your system package manager when working with Haskell, and utilize a tool like Stack instead. This will save you a lot of time avoiding arguments with Cabal.

1.1 Getting Your Environment Set Up

While Stack has largely supplanted any previous systems that were used for Haskell development, there is a still a chance that if you're reading this, you will be looking at an older, legacy project that hasn't adapted to it yet.

So, although we will be using Stack throughout most of this book, I feel obliged to at least mention how to get started with the older, Cabal-only approach; I implore you to explore Stack unless you're under explicit order to do it with Cabal.

With that being said, let us take a look at using Cabal.

1.1.1 Getting Set Up with Cabal

The first thing you will want to do when using Cabal is to make sure that you have downloaded the latest version of the Haskell Platform from `https://www.haskell.org/platform/`. This should guarantee an up-to-date version of Cabal is installed on your system, thus ensuring the best compatibility possible.

Once the Haskell Platform is installed, you should have ghc installed on your system, so let us begin the process of setting up a new project:

```
mkdir my_new_project
cd my_new_project
cabal sandbox init
```

The first two lines should be fairly self explanitory, but `cabal sandbox init` might be a bit confusing.

While it's not *strictly* required, sandboxing will greatly simplify your time using Cabal. If you do not use a sandbox, dependencies get installed into the global space of your machine, instead of being contained to your project. This might not be an issue if you only have one project, or your projects are each contained separate virtual machines or containers, but it can be a tremendous nuisance when working with multiple projects on the same machine, which may be required depending on your workflow.

Once your sandbox is created, you will want to run `cabal init`

```
tombert@hostname:~/my_new_project# cabal sandbox init

Config file path source is default config file.
Config file /root/.cabal/config not found.
Writing default configuration to /root/.cabal/config
Writing a default package environment file to
/root/my_new_project/cabal.sandbox.config
Creating a new sandbox at /root/my_new_project/.cabal-sandbox
root@6a2da51caa38:~/my_new_project# cabal init
Package name? [default: my-new-project] my_new_project
Couldn't parse my_new_project, please try again!
Package name? [default: my-new-project]
Package version? [default: 0.1.0.0]
Please choose a license:
 * 1) (none)
   2) GPL-2
   3) GPL-3
   4) LGPL-2.1
   5) LGPL-3
   6) AGPL-3
   7) BSD2
   8) BSD3
   9) MIT
  10) ISC
  11) MPL-2.0
  12) Apache-2.0
  13) PublicDomain
  14) AllRightsReserved
  15) Other (specify)
Your choice? [default: (none)]
Author name? Thomas Gebert
Maintainer email? thomas@gebert.sexy
Project homepage URL?
Project synopsis? A test project for everyone to enjoy
```

Generally, the defaults are fine for most projects, though you should pay special attention to the Haskell version (2010 or 98), as this can greatly influence what the language allows you to do. You also should pay attention to the location where the code exists (`src` in this case) and the name of the "main.hs" file.

As previously stated, there is not much reason now to run vanilla Cabal. It's not useless or horrible, and it's possible that if you need to work with an older project you will be forced to use it.

For new projects though, we will want to use Stack.

1.1.2 Getting Started with Stack

FPComplete's Stack is relatively easy to install. If you're on a Unix or Linux machine, installation is a fairly straightforward process of pasting a command into your terminal:

```
tombert@hostname: curl -sSL https://get.haskellstack.org/ | sh
```

1.1.2.1 Installation on Windows If you're on Windows, first consider using a Linux installation in a virtual machine or in a Linux server first. GHC tends to be a little out of date on Windows, and is usually at least one version behind. Of course, if you plan on building a Windows program, you will be forced to use the Windows version of Stack.

The installation on Windows isn't terribly hard, and involves a standard "next-next-next-finish" style of installer.

It should be noted that this book is going to assume that the user is using a Unix or Unix-Like environment, since the theme of this book is about network programming. While there are countless network programs written for Windows, Linux and other Unix-Like systems have become the clear-winner in the server space, and.

1.1.3 Using Stack

Starting a project with Stack is pretty simple. Open up a terminal and type:

```
tombert@hostname: stack new testproject
tombert@hostname: cd testproject
tombert@hostname: stack setup
```

You might notice that there is a `testproject.cabal` file in this folder. This is because Stack uses Cabal behind the scenes for dependency management.

1.2 Choosing an Editor

There exists a bit of religiousity in every programmer with text editors. People used to languages like Java or C# tend to prefer editors with a lot of features, such as IntelliJ or Visual Studio. Engineers who work in the weeds with C or C++ seem to tend gravitate a bit more towards a command-line interface utilizing Vim or Emacs.

For the former group, there now exists a couple of integrated environments that should be right up your alley. FPComplete has a Web-based editor that utilizes the excellent Docker platform to give handle project generation and testing. The editor has many of the bells and whistles that people like about Eclipse or IntelliJ.

As for the more minimalist groups, both Vim and Emacs have basic syntax-highlighting for Haskell out of the box, and that may be enough for you depending on your concerns. All the examples in this book are going to be built using the command-line, so it shouldn't be very difficult to adjust.

FPComplete has also built an editor backend called "Intero", which Vim and Emacs both have plugins to hook into. Intero allows you to get type information about variables and generate type signatures for functions.

Personally, I tend to prefer using the command-line editor called "NeoVim" (a fork of Vim), with no integrations outside of syntax highlighting. I use a terminal multiplexer called `tmux` to split my terminal in half so that I can have my editor on the top and a terminal on the bottom, and which allows me to quickly switch between a command line and an editor on the fly.

Whatever editor you choose, a pattern you will want to get into in Haskell is running things and experimenting in the interactive interpreter, known as `ghci` (accessible from `stack ghci` or `cabal repl`). Doing a full-on compilation of a Haskell program can take quite a long time, especially when your programs start getting big. As a result, it can be immensely useful, when you're just experimenting and making sure your logic and types are correct, to avoid the compilation when possible.

1.3 Hello World

In the time-honored tradition of nearly every programming book on the planet, let us write a simple program that writes "Hello World" to the screen. Asumming you generated your project with Stack (as shown in the example above), let's build and run the generated code to see what it gives us out of the box:

```
tombert@hostname: stack build
tombert@hostname: stack exec testproject-exe
someFunc
```

This generated stub appears to be an almost completely useless program; all it seems to do is simply print out the phrase `someFunc` to the screen.

We need to figure out what text we need to change to make our program say the standard "Hello world", so navigate to the `app/Main.hs` file in your favorite editor. You should see something like the following:

```
module Main where

import Lib

main :: IO ()
main = someFunc
```

Don't worry if you don't understand everything that's happening here; you have the entire book to learn that. Right now, all we need to understand is the `main`, like in most programming languages, is the entry-point to your application.

It appears to be that `main` is defined to be equal to `someFunc`. While it's not actually how the compiler works, it can be be useful to think of function declarations like this as a "copy" of the function on the right.

To find the definition of `someFunc`, let's point our editor over to the file `src/Lib.hs`. We should see something like this:

```
module Lib
    ( someFunc
```

```
    ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

It looks like `someFunc` is defined as `putStrLn "someFunc"`. This explains why the only thing logging to the console was `someFunc`.

Change the text in the string to `"Hello World"`, save the file, and build the program like we did above.

```
tombert@hostname: stack build
tombert@hostname: stack exec testproject-exe
Hello World
```

Congratulations, you are now officially a Haskell programmer.

While I don't expect you to fully understand this program yet, I would like to direct your attention to `someFunc`, where you might notice that it has a weird bit of text right above it (`someFunc :: IO ()`).

This is a function type signature, and while we will go into more detail later in the book about their significance, it might be useful to look at the `IO ()` part. This is the return type of the function. The `IO` tells the compiler that this functoin returns something with the `IO` monad, which signifies that this function has side-effects, while the `()` is called "unit", and is effectively the functional equivalent of `void`, and it indicates that we are not returning any kind of value from this function.

The reason that I bring all this up is to express this simple point: while a lot of problems actually become a lot when utilizing a pure-functional approach, other things (such as understanding how to write a simple "Hello World" program), can require a basic understanding of Category Theory and Type Theory.

While this can be overwhelmingly scary at first, and is where Haskell gets its reputation as a "hard" language, the more you use Haskell, the more you will start thinking about things in a functional style, and the more you will appreciate having to be explicit about side effects.

But we're getting ahead of ourselves, let's get started with the most fundamental part of functional programming: Functions.

CHAPTER 2

TYPES

In the beginning, computers could only be programmed in binary. 1's and 0's were painstakingly placed encoded by hand, and the pure logicians following the Boolean philosophy.

Once compilers hit the scene, these calculations could be done automatically. No longer were you manually constantly mapping out how many bytes each variable would uses, and instead describe the data in a more abstract sense, using `int`, or `float`.

The `C`s, `C++`s, and `Java`s of the world have kept this philosophy for types. The types exist as an optimization for the compiler, and largely serve little other purpose.

Haskell takes the types to a higher level, and makes them a useful, itegral part of the development.

For the most part, Haskell can deduce the types of your function.

Of course this can work, but there exists some issues:

- Variables initialized from the outside world end with the infamously difficult-to-debug `null` reference error.

CHAPTER 3

FUNCTIONS

3.1 What is a function?

Let me get the shocking truth out right away: most programing languages don't have functions.

Of course, nearly every language has a concept of a "function", and might even have a keyword designated for it. It might be useful in that programming language to think of these as "functions", but if we talk in a purely mathematical sense, they don't measure up to the major properties of functions.

Let us first look at Algebra, and examine the "vertical line test" for checking whether a graph could possibly represent a function. This simple procedure exists to show that for any single input value x, in the function $f(x)$, you will always get the same value out; that is to say, for any value x, there exists *at most one* value y which is the result of calling that function.

This may seem like a trivial point, but let's look at the JavaScript function `Date.now()`. It superficially looks like a function, and returns a value. Most of the documentation calls it a function, and in the way most programmers think of it, it might be considered a function in some kind of abstract sense.

But `Date.now()` doesn't pass the vertical line test. `Date.now()` takes no arguments, but returns a different result upon every execution. `Date.now()` is dependent on the computer's system clock, which is likely dependent on a service like NTP or `time.nist.gov`.

If I were to plot out the data mathematically, it would look something like this: (INSERT CHART)

Now, let's compare this to this mathematical function:

$$f(x) = x^2$$

If I supply the value 10 for x, The result will *always* be 100, no matter how many times I run it. It will be 100 whether or not a certain file exists on a hard drive. It will be 100 even if all of Google's servers break down. It will be 100 for always and forever simply because that is a universal property of functions.

3.1.1 Pure and Impure

In functional programming, we distinguish between predictable, mathematical functions and functions dependent on external factors as *pure* and *impure*.

Besides simple predictability, pure functions have several other advantages. By using them, it becomes easier to push more reliance onto the type system of the language, which allows the compiler to diagnose many most potential errors before.

Since impure functions require a basic understanding of monads, and since that can be a bit of a rabbit-hole in and of itself, we will focus on them in later chapters, and focus on pure functions in this chapter.

3.2 Defining Functions

Functions in Haskell tend to be fairly close to what you do in math. For example, if we wanted to define the $f(x) = x^2$ function from above, we'd write:

```
f x = x * x
```

Let's break each part down:

- `f` is the name of the function.
- The `x` before the `=` is a variable declaration. You can add as many variables as you would like, and you just need to separate them by spaces
- The `x * x` is the function body. Haskell is expression-based, not statement-based, so there is no need to explicitly call `return`

Haskell is a statically typed language, so you might be wondering where there isn't any type declaration. To understand why, open up a GHCI REPL by typing `stack ghci`. Type `f x = x * x` into the shell. Then type `:t f`. You should get the following output:

```
f :: Num a => a -> a
```

We will go into more detail about the specifics when we get to our chapter on Typeclasses, but what this type signature says is allow any type `a` that is a number. The `Num` is effectively a constraint that only allows numbers to be sent into the function, and this can all be deduced at compile-time, by inferring it out of the arguments demanded from the `*`.

3.2.0.1 Generic Types This type of generic programming should be familiar to anyone who has ever done anything with Java Generics or C++ templates.

While this kind of programming can be kind of cumbersome in imperative languages, it's largely automatic and simple in Haskell. Most of the time, the types can be deduced automatically, and GHC is very good about making the functions as generic as possible.

As a result, it's easy to make your Haskell programs incredibly modular and reusable, and you'll find that most of the time, your functions can have many uses.

Haskell's type system is so unique and interesting that they'll get their own chapter in the future, so for now just remember that the lowercase `a` is a symbol for a generic type.

3.3 Composing Functions

While many Haskell tutorials tend to wait until much later to start talks of function composition, I feel it's useful to learn basic composition early-on, if only to show the beauty of keeping your functions pure.

If you can again recall an example from Algebra, you might remember the circle operator defined like this:

$$f \circ g = f\left(g\left(x\right)\right)$$

That is to say, for the $\circ$ operator, we apply the value of the right function and send its result into the first function. The final result is a new function that effectively "wraps" this procedure from us.

This operation is called "composition", and it might seem so trival that it's not even worth talking about, and in it is actually a very simple operation.

However, this operation has one incredibly practical use; it's a mathematical way to glue computations together.

This should be familiar to anyone who has ever used the `|` (or "pipe") tool in Unix, and indeed the philosophy is very similar. By keeping functions small, and by focusing on having them do one thing and doing it well, and by giving a good framework to combine things together, we are able to largely eschew complexity.

What they might not have told you in algebra are a couple properties, such as:

$$(f \circ g) \circ h = f \circ (g \circ h) = f \circ g \circ h$$

$$f \circ g \neq g \circ f$$

Haskell has embraced this philosophy wholeheartedly, and has a special operator dedicated to it. Since we don't have a $\circ$ operator readily available on our keyboards, Haskell has opted for the simple `.` operator. For example, if we wanted a function that squares the result of our previous square function (`f x = x * x`), we could write it like this:

```
fourthed = f . f
```

If you type this into GHCI, then you should then be able to type

```
Main Lib> fourthed 2
```

```
16
```

This worked by first applying 2 to the right `f`. This would result in 2×2, which of course gives us 4. It then applies that 4 to the `f`, which results in 4×4, which gives us 16.

The brilliance of this is that both of the instances of `f` are completely decoupled and modular. If I instead wanted to subtract 1 from the final result instead of squaring it, I could write something like this:

```
subtractOne x = x - 1

squareThenSubtract = subtractOne . f
```

Or if I wanted to square the the result of that:

```
subtractOneSquare = f . subtractOne
```

Haskell composition follows the same properties as algebra in regards to precedence. For example:

```
squaresquaresquare = f . f . f
squaresquaresquare' = (f . f) . f
squaresquaresquare' = f . (f . f)
```

Will all give you the same output. This property can be useful for removing unnecessary parentheses, or adding parentheses for emphasis.

3.4 Types

Of course, sometimes you will need concrete types in your program.

Types in Haskell are designated as beginning with a capital letter, and the standard bunch of types you've seen in most other languages are included:

(Make a chapter about types)

- `Double` - A double-precision floating-point number.
- `Int` - A 32-bit integer.
- `Integer` - An arbitrary-precision integer. There is no risk of overflow, but these tend to perform worse than `Int`.
- `Char` - A single Unicode Character.
- `String` - A list of Characters.
  - Strings are a bit controversial in Haskell. We will have a whole section on them in a later chapter.