Haskell Lessons Documentation

Release 0.1

Justus Adam

Apr 12, 2017

CAPTERS:

CHAPTER

ONE

THE TOOLS WE WILL NEED

This fist lesson is all about the various tools we will use to develop Haskell code.

1.1 The compiler (GHC)

Haskell is a compiled language. As such you do not require any special tooling at runtime. However to develop and build Haskell projects you will require a compiler to generate an executable binary file from your code. Furthermore you will most likely require a library management tool, since the Haskell "base" library, which is bundeled with the compiler will most likely not be sufficient for most tasks[1].

There are several Haskell compilers out there, however very few are well maintained. As such the Glasgow Haskell Compiler, or GHC for short, has developed as the de-facto standard Haskell compiler. It is by far the most mature, stable and feature rich Haskell compiler. In this course we will use optional extensions of the Haskell language which not every compiler implements. I therefore highly recommend using the GHC as your Haskell compiler.

For more information about the various parts of the GHC see the compiler reference pages. There you will find information on compiler flags, the interactive prompt GHCi, including the debugger, profiling, and the GHC Haskell extensions. We will discuss all these topics in the future.

We will rarely interact directly with the compiler, as there are very nice build tools out there which we will make use of instead.

1.1.1 The interactive Interpreter (GHCi)

The GHC also supports the interpreted execution of code. For one this allows you to directly run a Haskell source file with the `runghc` or `runhaskell` program. Furthermore any standard installation of GHC includes a program in which you can interactively type Haskell code, inspect it and run it. The program is called GHCi (`ghci` is the executable name) which stands for "GHC interpreter". (ghci reference pages) GHCi is very simlar to programs like the python or ruby interpreter with the notable difference that the code you type is type checked, like normal Haskell programs, before it is executed. The GHCi also includes a debugger for Haskell code (similar to gdb) which we will study in a later chapter.

We first use GHCi to explore some Haskell code before we get started with source files.

Some notable ways to interact with GHCi are:

`<expr>` Simply submitting a Haskell expression evaluates the code and tries to print the result value.

`let <name> = <expr>` Binding the value of an expression to a name (we will learn about this *later*).

[1] This is one of the unfortunate things about Haskell, that the base library lacks many desirable things. Examples of good standard libraries would be those of python and go.

:? Probably the most important command. Displays the help menu. The help menu lists available commands (a selection of which will follow here) and what they do.

:browse <module> Displays the contents of the module with the entered name.

:type <expr> Prints the type of the expression `expr`.

:info <name> Displays information about the name, such as the source module, or the type if it is a function.

Some, but not all, of these commands also work in a shortened form (`:t` for `:type` for instance)

1.2 Documentation resources

The standard Haskell tool suite includes a tool called "haddock" which can be used to generate documentation for your source code from special in line comments. This documentation is available online for published packages.

1.2.1 Hackage

Hackage is the de-facto standard online Haskell package database. Anyone can make an account and start uploading their own Haskell packages. My username for instance is justus.

On hackage you may search for packages, browse the different versions available for each package, see a packages dependencies and also browse the generated haddock documentation.

1.2.2 Hoogle

Hoogle is a search engine for the hackage documentation. Whereas on hackage you may only search the database by package name on hoogle you can search the contents more directly by searching for function names, module names, package names and even *type signatures*.

1.2.3 Stackage

The *stackage* site, which host resources to be used with the tool `stack` functions similarly to a combination of Hackage and Hoogle. It hosts the documentation, including a Hoogle search, for each package snapshot. I therefore recommend to use stackage to browse haskell packages and documentation, unless the package you want information on is not on Stackage.

1.3 Editors

Any text editor is fine for Haskell development. Though it is desirable to have at least some Haskell source code highlighting. Many editors also offer extra features via one of the Haskell ide servers `ghc-mod` and `intero`.

I personally use *visual studio code*, becuase it is clean and fast and becuase I maintain its Haskell highlighting plugin and constantly improve it. However I have heard that the editor best supporting Haskell is supposedly emacs. Atom also has good Haskell support because of the atom-haskell group on github.

For those who wish to go hard on Haskell, there is a graphical editor written in Haskell itself, called leksah as well as a command line editor called yi. Also a special mention is to be given to Haskell for Mac a particularly beautiful graphical Haskell IDE with native stack support for OSX and tailored towards learning Haskell.

And lastly I want to mention ghcid. Its a very simple, command line based program which simply attempts to load your code into the interpreter an shows you the errors it encouters. It automatically refreshes whenever you save a source file. This gives you some very bare bones ide features. The big advantage is that, unlike the other ide programs, `ghcid` is incredibly reliable.

CHAPTER

TWO

FUNDAMENTALS OF THE HASKELL SYNTAX

This lesson contains gets us started with the basics of Haskell syntax. Haskell is an old language (older than Java) and also one with which people like to experiment. As a result a lot of extra syntax has accumulated in over the years. Some of it in regular use, some of it more obscure and not well known. Most of this extra syntax is hidden behind language extensions. We mway come to learn some of it in future lessons, however for now we will simply start with the ML style core of the Haskell syntax.

2.1 Comments

There are two types of comments in Haskell. **Line comments** start with a double dash `--` (must be followed by a space or word character) and extends to the end of the current line. **Block comments** start with the sequence `{-` and extend until the end sequence `-}`.

2.2 Type literals

In Haskell type literals always start with an uppercase letter. Examples from Haskell's base library are:

`Int` A fixed size integer

`Integer` An unbounded integer.

`Float` A floating point number.

`Char` A character.

`String` A string of characters.

`Bool` A boolean.

After that the allowed characters are word character, digits, the underscore `_` and the apostrope `'` (often called *"prime"*).[#type-operators] Therefore a name such as `Isn't_4_bool` is a valid type name.

In general Haskell is a type inferred language, meaning you rarley have to specify the type of a value or expression (although it is common practice to annotate top level types and values with type signatures). You can hovever annotate any value and expression you want with a type signature. The special operator `::` can be used to achieve this (see also next section for examples). This is particularly useful when chasing down the source of type errors as you can fix expressions to a certain type you expect them to be.

2.3 Value literals

Supported literals are:

Numbers `1`, `3.0` etc. These are however overloaded, meaning depending on the inferred type a literal `3` can be an `Int` or `Integer` for instance. If you wish to specify the type you can annotate the literal like so `(3 :: Int)`, `(3 :: Integer)` or `(3.5 :: Float)`, `(3.5 :: Double)`.

Characters Character literals are constructed by surrounding a character or excape sequence with single quotes.

`'a'`, `'H'`, `'5'`, `'.'`.

The escape character `\` is used to produce special values, such as the newline character (`'\n'`) For a literal `\` character use `'\\'`. For a literal `'` character use `'\''`.

Strings String literals are constructed by surrounding a sequence of characters or escape sequences with double quotes.

`"Hello World"`, `"Foo\nBar"`

The same escape sequences as for characters apply with addition of the excaped double quotes `\"`.[2]

Lists List literals are a sequence of comma separated values surrounded by square brackets.

`[1,2,3,4]`

All elements of a list must have the same type.

More on the list type in the next section.

Tuples Tuples are a sequence of comma separated values surrounded by parentheses.

`(5, "A string", 'c')`

Unlike lists the elements of a tuple can have different types.

More on the tuple type in the next section.

Note that there are not *special* literals for booleans in Haskell as they are just a regular data structure. The literals for `Bool` are `True` and `False`.

2.4 Bindings

Bindings (often also called variables) are names referring to pieces of data. It is similar to the concept of variables in other languages, however in Haskell bindings are **always** immutable. Since these ergo they are **not** "variable" (they cannot vary). This is why I prefer the name "binding" as it **binds** a value to an identifier, not a variable, as it cannot "vary".

Bindings must always start with a lowercase letter. Then, like the types, it may contain word characters, digits, the underscore and the apostrope.[4]

There are several ways to bind a value. The first one we will learn (because it is the way to bind values in GHCi) is called `let` with the concrete syntax `let name = value`.

```
let myInt = 5
let aBool = False
let someString = "Hello World"
```

[2] There is a language extension in GHC which allows overloading of strings (much like the numeric literals), see overloaded strings.

[4] The naming convention in Haskell is camel case. Meaning in each identifier (type variable, type or binding) all words composing the name are chained directly, with each new word starting with an upper case letter, except for the first word, who's case is determined by the syntax contstraints (upper case for types, lower case for type variables and bindings).

You can use `let` in GHCi to bind a value and then print it by simply entering the name again and pressing enter.[5] In the `let` construct you may also, optionally, specify a type signature for the binding.

```
let myInt :: Int
    myInt = 5
```

Note that the second occurrence of `myInt` must be properly indented. We will explore the indentation rules in more detail later.

2.5 `if` expressions

In Haskell `if` is not a statement, but an expression, meaning that it returns a value. Therefore `if` always has a type, and also always has an `else` case, which must return a value of the same type. For instance we can assign the result of `if` to a binding.

```
let aBool = False

let anInt = if aBool then 8 else 9
```

Parentheses are not required and one may write any expression on the branches and for the condition of an `if`.

2.6 Function Application

The syntax for applying functions to arguments in Haskell is the simplest imaginable. Its called *juxtaposition* or somethimes *prefix notation*. Meaning we simply write the function and follow it up by the arguments separated by whitespace. Optionally we can surround the whole construct with parentheses. This is especially useful when we need the result of a function call as an argument.

```
succ 5 == 6
takeDirectory "/etc/hosts" == "/etc"
elem (pred 6) [1..10]
not True == False
```

Haskell also supports binary operators. For instance the addition operator `(+)` and the equality operator `(==)`. Note that to apply the operator we use its bare form +, however if we mean a reference to the function we surround it *directly* with parentheses.

```
4 + 5 == 9
[1,2,3] ++ [4,5,6] == [1,2,3,4,5,6]

map (uncurry (+)) [(1,2), (4,5)]
```

Infix operators can also be used in the prefix notation by surrounding them with parentheses `(+)`. And prefix functions can also be used like infix operators by surrounding them with backticks ``4 `elem` [1..10]``.

Function application *always binds stronger* to its arguments than operator application. For operators users may define a prescedence in which they are applied. Thus `(+)` for instance is applied before `(==)`.

[5] Note that in GHCi, as in many Haskell constructs you may also **rebind** a binding. This may look like you have altered the binding, however this is not the case. It creates a wholly new binding, which simply shadows the older binding in the current scope. When the scope is exited the value stored for this name remains the old value. You will also know that it is a new binding by the fact that the new binding can have a different type than the old one.

footnotes

CHAPTER

THREE

EXERCISES

3.1 Tools

Install the mentioned Haskell tools.

3.1.1 Installing the compiler

- Method 1 - platform first

    1. Install the correct Haskell platform distribution for your system (for instance the package `cabal-install`).
    2. Install stack. Either from source with `cabal install stack` or by downloading it from the website.

    Advantages:

    With this approach you'll have the `ghc` and `ghci` as well as `cabal` and `haddock` commands directly available to you on the command line after installation.

    Disadvantages:

    You'll have to keep your installation of these tools up to date. If you later run `stack install haddock` to get a newer version of haddock it will shadow the previously globally installed tool.

- Method 2 - stack first

    1. Install stack from the website.
    2. Use the `stack setup` command on the command line to have stack install ghc for you.

    Advantages:

    No stray `ghc` etc. executables.

    Disadvantages:

    You have to separately install haddock with `stack install haddock` if you need haddock directly.

- Make sure the stack executable is on your `$PATH`.

    Stack can update itself (if you want to). In this case it'll always install the new version in some user-local binary directory (`~/.local/bin` on UNIX). Therefore you also have to ensure this directory is on your `$PATH`. This is also important as all other executables which you install with stack are placed in this directory (including the ones you wrote yourself).

3.1.2 Install some Haskell support for your favourite editor

You may ask for help for finding said support.

Also consider the website https://wiki.haskell.org/IDEs and https://wiki.haskell.org/Editors.

3.2 Basics

1. Open GHCi.
2. Browse the `Prelude` module.
3. Be overwhelmed with how much stuff is in there.
4. Open Hackage
5. Find the `base` library.
6. Find the `Prelude` module.
7. Browse the documentation of the prelude module on Hackage.
8. Play with the various functions from the `Prelude` module in the interpreter.

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search