Haskell Tutorial

A guide to Haskell basics

Ruben Saunders

Contents

1 Introduction

Haskell is a purely functional programming language. The following sections will give a brief overview of Haskell, and how to install it.

1.1 Language Overview

In Haskell, everything is a *pure* function - that is, they abide by the Mathematical definition of a function; they map inputs to a unique output.
Data is immutable, meaning that our data types cannot be changed in-place. Combined, this means that there a re few or no side-effects from functions, which make programming more simple.
Haskell is declarative, meaning that the program defines what the issue is, rather than simply giving an algorithm to solve a problem.
Functional programs are easier to verify as we can use maths to verify an algorithm.

1.2 Installation

Link: `https://www.haskell.org/ghcup/`
I used GHCup to install several components of the Haskell toolchain.

1.2.1 The Haskell Toolchain

The Haskell Toolchain consists of several useful tools for Haskell compilatio and development:

- **GHC** - the Glasgow Haskell Compiler;
- **cabal-install** - Cabal installation tool for managing Haskell software;
- **Stack** - a cross-platform proram for developing Haskell projects;
  - **Msys2** - provides a UNIX shell and environment which is necessary for executing configuration scripts.
- **haskell-language-server** - a language server which may be integrated into an IDE;

1.2.2 Install Command

The command to use on Windows (in a normal Powershell instance) is

```
Set-ExecutionPolicy Bypass -Scope Process -Force;[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; try { Invoke-Command -ScriptBlock ([ScriptBlock]::Create((Invoke-WebRequest https://www.haskell.org/ghcup/sh/bootstrap-haskell.ps1 -UseBasicParsing))) -ArgumentList $true } catch { Write-Error $_ }
```

2 Variables

Variables are name-bounded values. Variables in Haskell are immutable - they cannot be changed.

```
name = <value>
```

Variable types are *inferred*. To explicitly assign variables a type,

```
name :: Type
```

In GHCI, the type of a symbol may be retrieved by `:t name`.

2.1 Local Name Binding

Two methods are provided to bind a symbol to a value in a local scope: `let` and `where`. Each follow a "main" expression.

2.1.1 `let`

Syntax:

```
let
  <symbol> = <expr>
  <symbol2> = <expr2>
  ...
in
<main_expr>
```

For example, re-define `in_range` as follows:

```
in_range x min max =
  let
    in_lb = min <= x
    in_ub = max > x
  in
  in_lb && in_ub
```

2.1.2 `where`

Syntax:

```
<main_expr>
where
  <symbol> = <expr>
  <symbol2> = <expr2>
  ...
```

For example, re-define `in_range` as follows:

```
in_range x min max = in_lb && in_ub
  where
    in_lb = min <= x
    in_ub = max > x
```

3 Functions

Functions are defined to map a value from an input set - the *domain* - into an output set - *the co-domain*. Every output of the function is contained in a subset of the co-domain, called the *range*. **Pure** mathematical functions must be able to map every value from the domain, and each input value must map to only one output value.
In Haskell, occurences of functions are expanded into their RHS.

3.1 Function Definition

Functions are defined by providing its name, a list of arguments, and setting it equal to an expression.

`name arg1 arg2 ...argn = <expr>`

The arguments may be set to constants, or may be given a name to accept a variable value.
The type of a function is specified by an arrow (`->`) seperated list of its argument types and its return type:

$$\texttt{func :: type}_1 \texttt{ -> type}_2 \texttt{ -> ...-> type}_n \texttt{ -> type}_{\texttt{return}}$$

For an example, take a function which returns the sum of the elements of an array:

```haskell
sum :: [a] -> Int -- Takes an array of arbitrary type and returns an integer
sum [] = 0 -- Define the sum of an empty array to be zero
sum (h:t) = h + sum t -- Define the sum of an array to be the head plus the sum of the tail
```

3.1.1 Infix Functions

A good example of infix functions are operators such as `+`. Functions which take two arguments may be writen between the arguments instead.
For example, say we had `add a b = a + b`. Then `add 5 7` and ``5 `add`7`` are equivalent.
To reference the function defined by an operator i.e. `+`, surround it with parenthesis i.e. `(+)`.

3.1.2 Pattern Matching

Parameters of functions may be matched against a pattern. Note that order matters: more specific patterns should be defined first, with general cases last.

- Accept any value by using a symbol e.g. `f x = ...`.
- Accept a certain value e.g. `f 0 = ...`, `g [] = ...`
- List splicing using `(x:xs)` where `x` is the head, `xs` is the tail.
- Tuple unpacking e.g. `(Int,Int)` could be unpacked using `(x,y)`.
- Accept and extract a custom types. E.g., say we had `type Pos = (Int, Int)`, we would define `getX (x, y) = x`.
- Accept and extract a custom datatype. E.g., say we had `data Num = Zero | Succ Num`. We could then define `f Zero = ...` and `f (Succ n) = ...`.

Pattern matching may also be done using `case ... of ...` construct.

3.2 Function Application

A function is applied (called) to some arguments as follows:

`name arg1 arg2 ...argn`

For example, consider the function `in_range x min max = x >= min && x < max`, an implementation of $x \in [min, max)$.
Then, `in_range 3 0 5` would evaluate to True, but `in_range 5 0 5` would evaluate to False.

3.3 Recursion

Recursion is the process of a function calling itself. Recursion requires a *base case* to stop the function recursing indefinitely.
There are many ways to implement recursion, which will be demonstrated using the *factorial*, defined as

$$n! = n \cdot (n-1) \cdot \ldots \cdot 1 = \prod_{k=1}^{n} k$$

3.3.1 Defined Base Case

We can hard-code the case where the function is called with the base case:

```
fac 1 = 1
fac n = n * fib (n-1)
```

3.3.2 If-Else Expression

We can use the if-else expression:

`if <expr> then <ifTrue> else <ifFalse>`

For example,

```
fac n = if n <= 1 then 1 else n * fac (n-1)
```

3.3.3 Guards

Guards are similar to piece-wise functions.

```
<main_expr>
  | <expr> = <value>
  | <expr2> = <value2>
  ...
  | otherwise = <default_value>
```

Where `<expr>` is a boolean expression. If `<expr>` is matches, then `<value>` will be returned. If none is matched, the `otherwise` is returned.
For example,

```
fac n =
  | n <= 1     = 1
  | otherwise = n * fac (n-1)
```

3.3.4 Accumulators

In this example, we define an auxiliary function `aux` inside `fac` to calculate the the factorial

```
fac n = aux n 1
  where
    aux n acc
      | n <= 1     = acc
      | otherwise = aux (n-1) (n*acc)
```

This is called *tail recursion*. This is because the final result of `aux` is the result we want, meaning that it is much more memory efficient. A good compiler could even unwind this into a non-recursive imperative approach using a loop. (For more insight, see `https://www.youtube.com/watch?v=_JtPhF8MshA`.)
Normal recursion (using an above definition of `fac`):

```
fac 4
= 4 * fac 3
= 4 * (3 * fac 2)
= 4 * (3 * (2 * fac 1))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

Tail recursion (using the definition in this sub-section):

```
fac 4
= aux 3 4
= aux 2 12
= aux 1 24
= 24
```

3.4 Lambdas

Syntax:

`(\<args> -> <expr>)`

Some examples:

- `(\x -> x+1) 2` returns `3`.
- `(\x y z -> x+y+z) 1 2 3` returns `6`.

Lambdas may be bound to names.

3.5 Higher Order Functions

Higher order functions are functions that take other functions as arguments.
For example, a function which takes another function and applies it to an argument:

```
app :: (a -> b) -> a -> b
app f x = f x
```

A synonym of such a function is the dollar (\$) operator: `($) :: (a -> b) -> a -> b`.

3.5.1 Useful Higher Order Functions

- `map :: (a -> b) -> [a] -> [b]` applies a function to every element on an array.

  $$L' = \{f(x) : x \in L\}$$

  Example: `map (\x -> x^2) [1,2,3]` returns `[1,4,9]`.

- `filter :: (a -> Bool) -> [a] -> [a]` filters the list on a predicate.

  $$L' = \{x \in L : P(x)\}$$

  Example: `filter (\x -> mod x 2 == 0) [1,2,3,4,5]` returns `[2,4]`.

- `fold :: (a -> b -> b) -> b -> [a] -> b` processes a list with some function to produce a single value, starting at a given value.

  In Haskell, `fold` doesn't exist, but rather `foldr` and `foldl` which start folding at either end of the list respectively.

  $$\texttt{foldr}\ (op)\ a\ [x_1, x_2, \ldots, x_n] = x_1\ (op)\ x_2\ (op)\ \ldots\ (op)\ x_n\ (op)\ a$$

  $$\texttt{foldl}\ (op)\ a\ [x_1, x_2, \ldots, x_n] = a\ (op)\ x_n\ (op)\ x_{n-1}\ (op)\ \ldots\ (op)\ x_1$$

  Example: `foldr (+) 0 [1,2,3,4,5]` returns `15`.

3.6 Currying

The principle behind currying is that given

```
f :: a -> b -> c -> d
```

We could re-write this as

```
f' :: a -> (b -> (c -> d))
```

For example, one could define a function `add` in multiple ways:

```
add x y = x + y
add x = (\y -> x + y)
add = (\x -> (\y -> x + y))
```

3.7 Currying & Uncurrying

Let's illustrate these terms by defining functions,

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

3.7.1 Partial Function Application

Using the last definition of `add`, consider the result of `add 1`. This would be a new function; `add 1 :: Int -> Int`. This is known as a *section*.
A good example would be using `map`.

```
doubleList = map (\x -> x * 2)
```

3.8 Function Composition

Function composition is a way to combine functions. For this, we use the dot `(.)` operator.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Then, `(f.g)` is equivalent to `(\x -> f (g x))`.
For example, all three definitions of `descSort` are equivalent:

```
descSort = reverse . sort
descSort = (\x -> reverse (sort x))
descSort = reverse (sort x)
```

4 Types

Every expression in Haskell has a type. Types are inferred, even when given explicitly.
Types always begin with an UPPERCASE letter.

4.1 Variable Types

```
var :: type
```

4.1.1 Lists

To define a list of a `type`, one would write `[type]`. This may be nested.

4.2 Function Types

```
func :: type1 -> ... -> typeN -> ret_type
```

Where the function `func` takes n arguments of types `type1, ..., typeN` and returns `ret_type`.

4.2.1 Type Variables

Type variables may be used where any type would be permissable and must be lowercase. For example,

```
id :: a -> a
id x = x
```

This is called "parametric polymorphism".

4.3 Type Aliasing

This doesn't define a new datatype, but rather an alias for another type.

```
type Pos = type
```

A common example is using a tuple e.g. `type Pos = (Int, Int)`. The type name need not be stated in pattern matching.

4.4 Type Classes

Type classes may be used to restrict the types a polymorphic function may take. This is useful if we would like to use features in a polymorphic function that may only be available to certain types. For a type to be a member of a type class, it must implement all of the required methods.
To impose a constraint on variable `a` in a function `f`: `f :: (<TypeClass> a, ...) => ...`. This is called "ad-hoc polymorphism".

4.4.1 Definition

A type class definition begins with
`class <Name> <var> where`
Below is a list of function declarations. For a type to be a member of `<Name>`, it must implement all of these functions.

4.4.2 Implmentation

To define a new type which belongs to a type class: `instance <TypeClass> <TypeName> where`
Where below this is a list of function definitions.

4.4.3 Common Type Classes

Common type classes include:

- `Eq` – types which may be compared i.e. `(==)` is defined;
- `Num` – numeric types, gives us access to standard mathematical operations i.e. `(+), (-), (*), abs, ...` are defined;
- `Ord` – types which may be ordered, imposes a total ordering i.e. `(<), (>), (<=)` are defined;
- `Read` – types which may be converted from a string i.e. `read` is defined;
- `Show` – types which may be converted to a string i.e. `show` is defined;
- `Integral` – types which are integer-like i.e. `div, ...` is defined;
- `Floating` – types which are float-like i.e. `(/), ...` is defined;
- `Enum` – types which may be enumerated i.e. `succ, pred, ...` are defined;

4.4.4 Instances

Intances allow us to write functions which make use of type classes. Syntax:
`instance (<constraints>) => <typeClass> <value> where` followed by a list of function definitions.

4.4.5 Derivation

The `deriving` keyword can be used to automatically generate implementations for the given type class(es).
Syntax: `data <Name> = ...deriving (<Class1>, ...)`
Example: `data Shape = Circle Int | Rect Int Int deriving Show`. Then, `print (Circle 5)` → `Circle 5`.

4.5 Defining Datatypes

The data keyword is used to define a new datatype; unlike the above, these are entirely custom.
This may be done using the `data` keyword:

```
data Name = Constructor1 [<args>] | ...
```

where `<args>` are the *types* of each argument, not literals.
Constructors are either plain values, or functions which take `args` and return the datatype.
Data consturctors can include polymorphism by including type variables after `<Name>` e.g. `data Maybe a = Nothing | Just a`.

4.5.1 Examples

- `rock-paper-scissors.hs` – A basic example revolving around Rock-Paper-Scissors;
- `expr.hs` – A program to build and evaluate expressions;
- `tree.hs` – A representation of a tree structure;
- `nat-num.hs` – A definition of natural numbers using the successor function;

4.6 Records

Records allow data to be stored with an associated name.
Syntax:

```
data <Name> = <Name> { <field> :: <type>, ...}
```

This will automatically generate functions `<field> :: <Name> -> <type>` to extract said properties.
This has the side-effect that field names must be globally unique.

4.6.1 Multiple Constructors

Note that records may also have mutliple constructors,
`data Point = D2 { x :: Int, y :: Int } | D3 { x :: Int, y :: Int, z :: Int }`
Duplicate field names in this context is OK.
This will generate functions `x`, `y` and `z` all with the signature `x/y/z :: Point -> Int`. Both `x` and `y` will work on either `D2` or `D3`, but applying `z` to `D2` will throw an exception.
For an example, see `code/vector.hs`.

4.7 Maybe

The `Maybe` type is incredibly useful, as it can be used to represent the *absence* of a value. This is useful when our function is passed invalid data, for example.
If it defined as: `data Maybe a = Nothing | Just a`

4.7.1 Functions

Useful functions may be found inside `Data.Maybe`.

- `isJust :: Maybe a -> Bool` – returns if the passed `Maybe` is a `Just` value;
- `fromMaybe :: a -> Maybe a -> a` – returns the `Just` value if a value is present, else returns the default value;
- `fromJust :: Maybe a -> a` – returns the `Just` value, or throws an exception if recieved `Nothing`;
- `catMaybes :: [Maybe a] -> [a]` – returns a list containing all the `Just` values.

5 Collections

Haskell has two collections: lists, and tuples.

5.1 Lists

A *mutable* collection of elements of the *same type*. Every elements has an ordinal.
A list of type type has the given type signature

name :: [type]

5.1.1 Construction

A list may be greated by the following constructor:

- Using square brackets: [x_1, x_2, ..., x_n]
- Using the prepend opeator: $x_1 : x_2 : \ldots : x_n$:[]. With the syntax of x:list, it prepends x to the list list.

5.1.2 Pre-defined functions

Many pre-defined functions for lists are defined in the Data.List module.

General Functions These functions work on a list of any type, namely [a].

- head <list>. This function returns the head (x_1) of the list. Example: head [1,2,3] returns 1.
- tail <list>. This function returns the tail of the list. Example: tail [1,2,3] returns [2,3].
- length <list>. This function returns the length of the list. Example: length [1,2,3] returns 3.
- init <list>. This function returns the list without the last element. Example: init [1,2,3] returns [1,2].
- null <list>. This function returns whether the list is empty. Example: null [1,2,3] returns False.
- take <n> <list>. This function returns a list of the first n elements of the list. Example: take 2 [1,2,3,4,5] returns [1,2].
- drop <n> <list>. This function returns a list excluding the first n elements of the list. Example: drop 2 [1,2,3,4,5] returns [3,4,5].
- <list1> ++ <list2>. This function "append" returns a concatenation of both lists. Example: [1,2] ++ [3,4] returns [1,2,3,4].

Boolean Functions These functions are of the type fn :: [Bool] -> Bool

- and <list>. This functions returns True if every elements in <list> is True.
- or <list>. This functions returns True if at least one element in <list> is True.

5.1.3 List Comprehension

List comprehension can be used to transform one or more lists according to a predicate. Syntax:

```
[ <gen> | <elem> <- <list>, ..., <guard>, ...]
```

Where

- <elem> <- <list> is called a *generator* – it binds each value from <list> to <elem> in turn so that they may be used. There may be multiple generators, in which case they will be worked through left-to-right.

- `<guard>` is a statement which returns a `Bool`. If false, the current bound value(s) will not be output.

Examples:

- `[ 2*x | x <- [1,2,3] ]` generates `[2,4,6]`
- `[ x^2 | x <- [1,2,3], x > 1 ]` generates `[4,9]`
- `[ (x,y) | x <- [1,2,3], y <- ['a','b'] ]` generates
  `[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]`

5.1.4 Ranges

Generate ranges (arithmetic sequences) in Haskell using the ellipse ..:

```
[<start>, [<step>] ..  [<end>]]
```

Where

- `step` is optional, and default to 1 e.g. `[1..5] = [1,2,3,4,5]`, `[1,3..5] = [1,3,5]`
- `<end>` may be omitted to generate an infinite list e.g. `[1 ..]  = [1,2,3,...]`.

5.2 Tuples

An *immutable* collection of elements of *different types*.
A tuple has the signature

$$\texttt{(x,y,...)} \;\; \texttt{::} \;\; \texttt{(type}_x\texttt{, type}_y\texttt{, ...)}$$

6 Modules

In Haskell, each file is a module. Hence, each file (other than the entry file) must begin with a module declaration:

`module` *filename* `[(n1, n2, ...)]` `where`

By default, every symbol is exported. If `(n1, n2, ...)` is included, only symbols `n1`, `n2` etc. are exported.

6.1 Importing

To import a module, use an import statement:

`import` *module* `[(n1, n2, ...)]`

By default, every symbol exported by `module` is imported. If `(n1, n2, ...)` is included, only symbols `n1`, `n2` etc. are imported.
To import `Animals.hs` one would write `import Animals`. To import `Farm/Tractor.hs` one would write `import Farm.Tractor`.
Once imported, symbols may be used freely. For example, if the function `double` is imported, to reference it we would write `double`. However, if two different definitions for `double` are imported, we must use its full name e.g. `Module.double`.

6.1.1 Qualified Imports

Syntax:

`import qualified` *module* `[(...)]`

This forces the module name to precede any symbols imported. In the example above, `Module.double` **must** be used.

6.1.2 Aliased Imports

Syntax:

`import` *module* `[(...)]` `as` *alias*

Using the above example, now, instead of writing `Module.double` one would now write `Alias.double`.

6.1.3 Import Hiding

The `hiding` keyword can be used to omit imports. E.g. `import Prelude hiding (map)` would import every function from `Prelude` but omit `map`.

7 I/O

I/O produces an issue with Haskell as I/O functions aren't *pure*.

7.1 The `IO` Type

All I/O functions in Haskell have the following type: `IO <value>`.
This special type holds a given I/O action. When the `IO` value is used, the stored action will be carried out, and `IO <value>` is returned.
For example, in GHCI

```
> hi = putStrLn "Hello, World!"
> hw
Hello, World!
```

Notice how nothing was outputted until the `IO` value was used. Note that `hw` may be used mutliple times.

7.2 Input

- `getLine :: IO String` – retrieves a line of input from STDIN;

7.3 Output

- `putStr :: String -> IO ()` – puts the given string to STDOUT;
- `putStrLn :: String -> IO ()` – puts the given string to STDOUT on a new line;

7.4 Extracting `<value>`

`IO` is a *monad*, and should be extracted as such.

```
greet :: IO ()
greet = do
  putStrLn "What's your name?  "
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

You can only extract values from `IO` inside of another `IO` action.
For a more complex example, see `code/IO.hs`.

8 Monads

With the signature of `Monad m => m a`, a monad is simply a wrapper around another type. A monad implements the following functions (only `(>>=)` is required): `bind` (`(<<=)`); `then` (`(>>)`); `return`; and `fail`.

8.1 Bind

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

This function takes a monad and a function which takes a raw value and returns a new monad, and returns another new monad.
When implemented, then, we may vary the action taken depending on the value of the provided monad, such as returning a default value – this is what `(>>=)` does with `Maybe`, as shown below:

8.1.1 Example with `Maybe`

```
add :: Num a => Maybe a -> Maybe a -> Maybe a
add mx my = mx >>= (\x -> my >>= (\y -> Just (x + y)))
```

Then addition works as expected:

```
> add (Just 1) (Just 2)
Just 3
```

But if either `mx` or `my` is `Nothing`, `(>>=)` skips the function and returns a default of `Nothing`, propagating the "error".

```
> add Nothing (Just 2)
Nothing
```

8.2 Then

```
(>>) :: Monad m => m a -> m b -> m a
```

This function discards the second monad given to it. `m >> n` is equivalent to `m >>= \_ -> n`.
"Then" can be though of wanting to carry out an action but not caring what the result is.

8.3 Return

```
return :: Monad m => a -> m a
```

Return wraps a monad around a raw value.
Using the example from the Bind section, we could substitute the explicit `Just` with the more general `return`. Now, this would theoretically work with any appropriately-defined monad.

```
add mx my = mx >>= (\x -> my >>= (\y -> return (x + y)))
```

8.4 Fail

```
fail :: Monad m => String -> m a
```

Fail is intended to be called when something goes wrong. The default implementation is to call `error` (i.e. error out of the program), but it may be implemented so that certain errors may be handled and return an appropriate monad as a response.

8.5 "do" Syntax

Chaining together applications of `(>>=)`, `(>>)` and lambda functions can get tedious; that's where the syntactic sugar "do" expression comes in.
The "statements" inside of `do` are executed in order, and if one "statement" fails this will be propagated through.

8.5.1 Bind

We can substitute the "bind" construct

```
m >>= (\x -> ...)
```

with

```
do
  x <- m
  ...
```

8.5.2 Then

We can substitute the "then" construct

```
m >> ...
```

with

```
do
  m
  ...
```

8.5.3 Example

Translating the above `add` function into `do`-syntax:

```
add mx my = do
  x <- mx
  y <- yx
  return (x + y)
```

8.6 Monad Laws

Defined monads have the following properties. Any monad implemented must abide by these also.

- **Left identity** – `return a >>= k` ≡ `k a` where `a` is any value, `k` is a function;
- **Right identity** – `m >>= return` ≡ `m` where `m` is a monad;
- **Associativity** – `m >>= (\x -> k x >>= h)` ≡ `(m >>= k) >>= h` where `m` is a monad, `k`,`m` are functions.