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 retriesed 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:

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

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

```
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.
- 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))
```

3.6.1 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)
```

3.7 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 Collections

Haskell has two collections: lists, and tuples.

4.1 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]
```

4.1.1 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`.

4.1.2 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`.

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`.

4.1.3 List Comprehension

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

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

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')]`

4.2 Tuples

An *immutable* collection of elements of *different types*.

A tuple has the signature

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

5 Types

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

5.1 Variable Types

```
var :: type
```

5.1.1 Lists

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

5.2 Function Types

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

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

5.2.1 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
```

5.3 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.

5.4 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.

5.4.1 Examples

- `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;

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`.