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:

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

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.

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
```

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

(x,y) :: (type$_x$, type$_y$)