Introduction to the Haskell Programming Language

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

October 11, 2021

Table of Contents

Table of Contents

NOTE: This is NOT a complete introduction to Haskell; this is more of a high-level overview of the language. I highly recommend reading the book *Learn Yourself a Haskell for Great Good* for a nice tutorial of the language.

Overview

Haskell is a statically-typed functional programming language, which means that types are checked at compile-time instead of run-time.

Haskell is more “straight-jacketed” than Scheme, partly due to its being statically-typed, and also partly due to conventions that more strongly enforce functional programming style.

When running standalone programs, they need to start with the line

```
module Main where
```

and they need to have `main` defined.

Hello World Example in Haskell

```
-- hello.hs -- Hello World in Haskell
module Main where
main = putStrLn "Hello World!"
```

We can compile the program by running `ghc hello.hs` at the command line. We can run it by running `./hello`.

There is also an interactive REPL that can be accessed by running the `ghci` command. When using the REPL, you don't need to provide `module Main where`, and you don't need to define `main`.

Defining Functions in Haskell

Below are some examples of function definitions in Haskell:

```
-- Given a number, double it.
double x = 2 * x
-- Given two numbers, return the larger one.
larger x y = if x > y then x else y

-- Usage Examples
double 10
larger 16 7
```

Note that by default functions are written in prefix notation, without parentheses. Parentheses are used for explicitly defining arguments (e.g., `double (double 10)` is valid Haskell and returns 40, but `double double 10` is invalid) and for specifying precedence.

Any binary function (i.e., a function with two arguments) can be made infix by using tick quotes `.

```
-- Usage example
16 `larger` 7 -- equivalent of (larger 16 7)
```

Note that Haskell has many built-in infix binary functions that don't require tick quotes, such as those for arithmetic and Boolean operators.

Basic Types in Haskell

Haskell has built-in support for basic types such as integers, floating-point numbers, Boolean values, and characters. Haskell also has built-in support for lists and tuples. Strings in Haskell are lists of characters. Unlike Scheme, lists and tuples are homogeneously-typed in Haskell.

Lists in Haskell

The syntax for lists in Haskell is more conventional

```
myList = [1, 2, 3, 4, 5]
names = ["Ash", "Misty", "Brock"]
```

We can concatenate lists using the ++ operator.

```
[1, 2, 3] ++ [4, 5] -- returns [1, 2, 3, 4, 5]
"My name " ++ "is Michael" -- returns "My name is Michael"
```

Lists in Haskell

We can get the element at a particular index by using the `!!` operator.

```
[1, 2, 3, 4, 5] !! 2 -- returns 3
"This is a test" !! 4 -- returns ' ' (a space)
a = [2, 4, 6, 8, 10]
a !! 2 -- returns 6
```

Lists in Haskell

Other functions that operate on lists include:

- `head list` – Retrieves the first element of a list; analogous to Scheme's `car`.
- `tail list` – Retrieves all but the first element of the list; analogous to Scheme's `cdr`.
- `last list` – Retrieves the last element of the list
- `init list` – Retrieves all but the last element of the list
- `length list` – Retrieves the length of the list
- `take n list` – Retrieves the first *n* elements of the list
- `drop n list` – Retrieves a new list where the first *n* elements of the given list are omitted.
- `elem x list` – Checks to see whether *x* is in the list

Ranges

Lists in Haskell have built-in support for ranges.

`[a..b]` indicates a list with elements ranging from `a` to `b`, inclusive and without skipping steps.

For example, `[1..5]` is equivalent to `[1, 2, 3, 4, 5]`. This works for letters: `['a'..'z']` results in a string containing all lowercase letters in the alphabet in order.

Ranges can be descending; for example, `[5..1]` is equivalent to `[5, 4, 3, 2, 1]`. To represent ranges with skips, specify the second element. For example, `[1, 3..10]` is equivalent to `[1, 3, 5, 7, 9]`.

List Comprehensions

For more complex list constructions, we can use list comprehensions.

For example, if we want a list of the square roots of all integers between 1 and 10, we can run `[sqrt x | x <- [1..10]]`. This should be reminiscent of `map` in Scheme.

We can also perform filtering on the results. For example, if we want all even numbers between 1 and 100, we can run `[x | x <- [1..100], even]`. We use a comma to specify a filter.

Fizzbuzz demo in Haskell

Functions and Type Specifications

Thus far we have not annotated our custom functions with type specifications. For simple functions, type specifications are optional.

Functions and Type Specifications

```
-- Without type specifications
double x = x * 2
multThree x y z = x * y * z

-- With type specifications
double :: Integer -> Integer
double x = x * 2

multThree :: Integer -> Integer -> Integer -> Integer
multThree x y z = x * y * z
```

Anatomy of a Type Specification

```
double ::  Integer -> Integer
```

- The text preceding `::` specifies the name of the function which type is being defined.
- The text after `::` specifies the types of the input and output.
- `Integer -> Integer` indicates a function that inputs a value of `Integer` type and outputs a value of `Integer` type.
- Note that all type names in Haskell start with a capital letter.

Now, what is going on here?

```
multThree :: Integer -> Integer -> Integer -> Integer
multThree x y z = x * y * z
```

Now, what is going on here?

```
multThree :: Integer -> Integer -> Integer -> Integer
multThree x y z = x * y * z
```

Let's recall the lambda calculus, where functions like $\lambda x.x$ only have one parameter. In the lambda calculus, we can support multi-parameter functions by treating them as the repeated application of single-parameter functions that return closures. This is known as *currying*, named after logician Haskell Curry.

- When currying the function `multThree`, the function application `multThree a b c` is treated as `((multThree a) b) c`.
- In fact, `multThree a b c` is syntactic sugar for `((multThree a) b) c`.
- When performing `multThree a`, it returns a closure that accepts argument `b`. When calling that closure, it returns another closure that accepts argument `c`. When calling that closure, it returns the answer of `multThree`.

We can express this in terms of Scheme syntax:

```
(define (multThree a)
  (lambda (b)
    (lambda (c) (* a b c))))
```

This is the same concept in Haskell syntax:

```
multThree a = \b -> (\c -> a * b * c)
```

Note that `\x -> x` in Haskell is equivalent to `(lambda (x) x)` in Scheme. The above can be called as `((multThree 2) 3) 4`, but it can also be called as `multThree 2 3 4` due to the fact that all multi-parameter functions are curried in Haskell.

Thus, when we are specifying the type of a multi-parameter function:

```
multThree :: Integer -> Integer -> Integer -> Integer
multThree a b c = a * b * c
```

Type Variables

What happens in situations where we want to create more generically-defined functions?
For example, a function computing the length of a list should not care about the types of the elements in the list.

In this situation, we can use a *type variable* to supply a generic type. Type variables are lowercase.

```
-- input is a list of generic type
length :: [a] -> Integer
```

Table of Contents

Topics to Be Covered Next Lecture

- Type Classes
- Algebraic Data Types
- Pattern Matching
- Monads