Intro to Haskell Notes: Part 5

Adrian Brasoveanu[*]

October 5, 2013

Contents

1 Curried functions and related issues

Haskell functions can take functions as arguments and return functions as values. A function that does either of those is called a higher order function because it is defined over a domain or range that is itself functional.

Higher order functions aren't just a part of the Haskell experience, they pretty much *are* the Haskell experience: if you want to define computations by defining what stuff is instead of defining steps that change some state and maybe looping them, higher order functions are indispensable.

The exact same thing can be said about the formal semantics of natural language in the Montagovian tradition: higher order functions are basically the Montagovian formal semantics 'experience'. And the trade-off between 'loops' (i.e., nested quantifiers, e.g, an existential in the scope of a universal) and higher-order functions (e.g., a choice function instead of an existential quantifier) is pervasive throughout advanced research in formal semantics.

Higher order functions are a really powerful way of solving problems and thinking about complex computations, whether these computations are Haskell programs or natural language meanings.

1.1 Curried functions

Every function in Haskell officially takes only one argument. So how is it possible that we defined and used several functions that take more than one argument so far? We curried them.

[*]Based primarily on *Learn You a Haskell for Great Good!* by Miran Lipovača, `http://learnyouahaskell.com/`.

What does that mean? Take the *max* function. It looks like it takes two arguments and returns the one that's bigger. But in fact, doing *max* 4 5 first creates a function that takes an argument and returns either 4 or that argument, depending on which one is bigger. Then, 5 is applied to that function and that function produces our desired result.

That is, the following two calls are equivalent:

```
ghci 1> max 4 5
  5
```

```
ghci 2> (max 4) 5
  5
```

Putting a space between two things is simply function application. The space is like an operator and it has the highest precedence.

Let's examine the type of *max*:

```
ghci 3> :t max
  max :: Ord a ⇒ a → a → a
```

It's $max :: (Ord\ a) \Rightarrow a \to a \to a$, which can also be written as $max :: (Ord\ a) \Rightarrow a \to (a \to a)$. That can be read as: *max* takes an *a* and returns a function that takes an *a* and returns an *a*.

The *max* function is curried just like we curried the meaning of transitive verbs in English, or the meanings of the sentential connectives *and*, *or* and *if*.

1.2 Partially applied functions

How is this useful? For natural language semantics, it's useful because now we have a semantics that matches the English syntax very nicely. But this is a consequence of a more general benefit that we get with currying: currying enables new modes of meaning and program composition.

If we call a function with too few arguments, we get back a partially applied function, i.e., a function that takes as many arguments as we left out. That is, we can generate functions on the fly and make full use of them: apply them to data in a separate function call, pass them as arguments to other functions, compose them etc.

We make use of this in Montagovian semantics all the time: the essence of composing natural language meanings in a syntax-driven way is passing around partially applied functions.

Take a look at this simple function:

```
ghci 4> let {multThreeInts :: Int → Int → Int → Int;
         multThreeInts x y z = x * y * z}
```

Take a close look at its type:

```
ghci 5> :t multThreeInts
  multThreeInts :: Int → Int → Int → Int
```

What happens when we do *multThreeInts* 3 5 9, or equivalently ((*multThreeInts* 3) 5) 9?

```
ghci 6> multThreeInts 3 5 9
  135
```

- first, 3 is applied to *multThreeInts*, because they're separated by a space; that creates a function of type $Int \to Int \to Int$, namely $((multThreeInts\ 3)\ y)\ z = 3 * y * z$

```
ghci 7> :t (multThreeInts 3)
  (multThreeInts 3) :: Int → Int → Int
```

- note how applying a curried function to an appropriate argument 'cancels' an *Int* in the type signature of *multThreeInts* in the same way that we simplify fractions by canceling denominators when we do multiplication, e.g., $\frac{5}{3} \cdot 3 = \frac{5}{\cancel{3}} \cdot \cancel{3} = 5$; see for example the discussion of Ajdukiewicz fractions in these lecture notes on categorial grammars by Christian Retoré
- then 5 is passed to this new function, which creates another function of type $Int \to Int$, namely $((multThreeInts\ 3)\ 5)\ z = 3 * 5 * z$, which can be rewritten more concisely as $((multThreeInts\ 3)\ 5)\ z = 15 * z$

```
ghci 8> :t (multThreeInts 3 5)
  (multThreeInts 3 5) :: Int → Int
```

- finally, 9 is applied to this function and the result is an object of type *Int*, namely $((multThreeInts\ 3)\ 5)\ 9 = 15 * 9 = 135$

```
ghci 9> :t (multThreeInts 3 5 9)
  (multThreeInts 3 5 9) :: Int
```

Recall that the type of this function can also be written as $multThreeInts :: Int \to (Int \to (Int \to Int))$, which makes the currying fully explicit. The type before each occurrence of $\to$ is the type of the argument that the function takes, and the type after the same occurrence of $\to$ is what the function returns.

So, to recap:

- our function takes an *Int* and returns a function of type $Int \to (Int \to Int)$
- similarly, this function takes an *Int* and returns a function of type $Int \to Int$
- finally, this function just takes an *Int* and returns an *Int*.

1.2.1 Another example along the same lines

Here's one more example:

```
ghci 10> let multTwoIntsWith9 = multThreeInts 9
ghci 11> :t multTwoIntsWith9
    multTwoIntsWith9 :: Int → Int → Int
```

```
ghci 12> let multOneIntWith18 = multTwoIntsWith9 2
```

```
ghci 13> :t multOneIntWith18
    multOneIntWith18 :: Int → Int
```

```
ghci 14> multOneIntWith18 10
    180
```

And this final function call is equivalent to the more direct one:

```
ghci 15> multThreeInts 9 2 10
    180
```

By calling functions with ‘too few’ arguments, we’re creating new *bona fide* functions on the fly.

1.2.2 A final practice example (with an η-reduction twist)

Let’s work through a final example. Suppose we want to create a function that takes a number and compares it to 100. We could write it like this:

```
ghci 16> let {compareWith100 :: (Num a, Ord a) ⇒ a → Ordering;
          compareWith100 x = compare 100 x}
```

```
ghci 17> :t compareWith100
    compareWith100 :: (Num a, Ord a) ⇒ a → Ordering
```

For example, if we call it with 99, it returns *GT*:

```
ghci 18> compareWith100 99
    GT
```

But x appears on both sides of the equation in the above function definition. So we can rewrite it more simply as:

```
ghci 19> let {compareWith100 :: (Num a, Ord a) ⇒ a → Ordering;
          compareWith100 = compare 100}
```

```
ghci 20>  :t compareWith100
    compareWith100 :: (Num a, Ord a) ⇒ a → Ordering
```

The type declaration is the same (and the function is the same) because *compare* 100 returns a function: *compare* has a type of $(Ord\ a) \Rightarrow a \rightarrow (a \rightarrow Ordering)$ and calling it with 100 returns a function of type $(Num\ a, Ord\ a) \Rightarrow a \rightarrow Ordering$.

Incidentally, the additional *Num* class constraint sneaks up there because 100 is also part of the *Num* typeclass.

1.3 Sections

Up until now, we saw prefix functions being partially applied. But infix functions can also be partially applied by using sections.

We section an infix function by supplying an argument on only one side and putting the function and the argument in parentheses. For example:

```
ghci 21> let {divideBy10 :: (Floating a) ⇒ a → a;
          divideBy10 = (/10)}
```

```
ghci 22>  :t divideBy10
    divideBy10 :: Floating a ⇒ a → a
```

So *divideBy10* 200 is equivalent to (/10) 200, which is equivalent to the more customary 200 / 10:

```
ghci 23> divideBy10 200
    20.0
```

And here's another example of using sections: a function that checks if a character supplied to it is an uppercase letter:

```
ghci 24> let {isUpperLetter :: Char → Bool;
          isUpperLetter = (∈ ['A'..'Z'])}
```

```
ghci 25>  :t isUpperLetter
    isUpperLetter :: Char → Bool
```

```
ghci 26> isUpperLetter 'E'
    True
```

```
ghci 27> isUpperLetter 'e'
    False
```

1.3.1 Sections and negative numbers

The only special thing about sections is using $-$. From the definition of sections, (-4) would result in a function that takes a number and subtracts 4 from it. However, for convenience, (-4) denotes the negative integer minus four.

If you want to make a function that subtracts 4 from the number it gets as an argument, partially apply the *subtract* function like so:

```
ghci 28>  :t (subtract 4)
    (subtract 4) :: Num a ⇒ a → a
```

Or in infix notation:

```
ghci 29>  :t (`subtract`4)
    (`subtract`4) :: Num a ⇒ a → a
```

For example:

```
ghci 30>  subtract 4 9
    5
```

```
ghci 31>  4 `subtract` 9
    5
```

2 Printing functions in `ghci`

What happens if we try to just do *multThreeInts* 3 4 in `ghci` instead of binding it to a name with a **let** or passing it to another function? `ghci` will tell us that the expression is a function of type $a \to a$, but it doesn't know how to print it to the screen.

```
ghci 32>  multThreeInts 3 4
```

Functions aren't instances of the *Show* typeclass, so we can't get a string representation of a function.

When we do $1 + 1$ at the prompt, for example, `ghci` first evaluates that expression to 2 and then calls *show* on 2 to get a textual representation of that number. The textual representation of 2 is just the string `"2"`, and this gets printed on the screen.

3 Higher order functions

Up until now, we saw functions that return other functions when they are partially applied. But Haskell functions can also take functions as arguments. Such functions are truly higher order functions because they operate over other functions and not over first order objects like integers, characters etc.

All the functions that we defined above were underlyingly first order functions. This might have been less obvious because the functions were curried, so they had other functions 'embedded' inside of them. But if we look beyond this cosmetically higher order appearance, we see that all those functions operate over first order domains.

The introduction of truly higher order functions in this section is really taking things 'to the next level'. These functions are an essential ingredient in both natural language semantics and functional programming: they enable us to extract and encapsulate general properties of (classes of) meanings / programs and also, general patterns of meaning / program combination.

There are many examples of higher order functions in natural language semantics: reflexives (at least under certain analyses), quantifiers, determiners, modals, modifiers (e.g., adjectives, adverbs), wh-words that form relative clauses and questions etc.

3.1 A simple example

To illustrate higher order functions, we're going to make a function that takes a function and then applies it twice to something.

```
ghci 33> let {applyTwice :: (a → a) → a → a;
              applyTwice f x = f (f x)}
```

First of all, notice the type declaration:

```
ghci 34>  :t applyTwice
    applyTwice :: (a → a) → a → a
```

We didn't need parentheses before because → is naturally right-associative. But they're mandatory here:

- the parentheses in the type declaration *applyTwice* :: $(a \to a) \to a \to a$ indicate that the first argument of *applyTwice* is a function from objects of type a to objects of the same type a
- so *applyTwice* is a 'bona fide' higher order function; its higher order type is not simply a result of currying a function or relation that is actually first order

The second argument of *applyTwice* is also something of type a, since this second argument ends up being fed into the first (functional) argument. Finally, the return value of *applyTwice* is also of type a.

Note that a is a type variable, it can be instantiated as *Int*, *String* etc. But once we fix it for one of the arguments, all the other occurences of a have to have the same type. Here are some examples:

```
ghci 35> applyTwice succ 10
    12
```

```
ghci 36> applyTwice (+3) 10
    16
```

```
ghci 37> applyTwice (++" HAHA") "HEY"
    "HEY HAHA HAHA"
```

```
ghci 38> applyTwice ("HAHA " ++) "HEY"
    "HAHA HAHA HEY"
```

```
ghci 39> applyTwice (multThreeInts 2 2) 9
    144
```

```
ghci 40> applyTwice (3:) [1]
    [3,3,1]
```

We could read the type declaration of *applyTwice* the curried way like we did above, but it's sometimes easier to just say that *applyTwice* takes two arguments, a function of type $a \to a$ and an object of type a, and returns an object of type a.

From now on, we'll often say that functions take several arguments despite the fact that each function in Haskell actually takes only one argument at a time and returns partially applied functions. For example, we'll say that a function of type $a \to a \to a$ takes two arguments of type a even though we know what's really going on under the hood.

We'll use similar 'abbreviations' when we talk about meanings for natural language expressions: they will all be curried, but we'll often talk as if they aren't if that's somehow more intuitive.

But for *applyTwice*, reading the type signature the curried way is actually better because it captures the essence of the function: its type is $(a \to a) \to a \to a$ or more explicitly, $(a \to a) \to (a \to a)$. This last way of writing the type signature clearly indicates that *applyTwice* takes a function as its argument and returns a function as its value. And the subsequent definition of *applyTwice* tells us that the function it returns is just a double application of the function it receives as input.

3.2 The versatility of higher order functions: implementing and using *zipWith*

Now we're going to use higher order programming to implement a really useful function that's in the standard library: *zipWith*.

It takes a function and two lists as arguments and then joins the two lists by applying the function to pairs of elements drawn from the two lists:

```
ghci 41> let {zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c];
              zipWith' _ [] _ = [];
              zipWith' _ _ [] = [];
              zipWith' f (x : xs) (y : ys) = f x y : zipWith' f xs ys}
```

Look at the type declaration:

```
ghci 42> :t zipWith'
    zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
```

The first argument is a function that takes two things and produces a third thing. They don't have to be of the same type, but they can.

The second and third arguments are lists and the result is also a list:

- the second argument has to be a list of entities of type a because the joining function takes entities of type a as its first argument

- the third argument of *zipWith′* has to be a list of entities of type *b* because the second argument of the joining function is of type *b*
- and the result of *zipWith′* is a list of entities of type *c* because the result of the joining function is of type *c*

The action in the *zipWith′* function is pretty similar to the normal *zip*. The edge conditions are the same except there's an extra argument, the joining function. But that argument doesn't matter in the edge conditions, so we just use a _ for it.

Finally, the function body in the last pattern is also similar to *zip*, only instead of forming a pair (x, y), we apply the joining function to the two elements, i.e., $f\ x\ y$.

For ease of reference, the implementation of *zip′* from the previous set of lecture notes is provided below. Compare it with the implementation of *zipWith′* above.

(1) $zip' :: [a] \to [b] \to [(a, b)]$
$zip'\ \_\ [\,] = [\,]$
$zip'\ [\,]\ \_ = [\,]$
$zip'\ (x : xs)\ (y : ys) = (x, y) : zip'\ xs\ ys$

A single higher order function can be used for a wide variety of tasks if it's general enough. Here's a little demonstration of all the different things our *zipWith′* function can do:

```
ghci 43> zipWith′ (+) [4,2,5,6] [2,6,2,3]
    [6,8,7,9]
```

```
ghci 44> zipWith′ max [6,3,2,1] [7,3,1,5]
    [7,3,2,5]
```

```
ghci 45> zipWith′ (++) ["foo ","bar ","baz "] ["fighters","hoppers","aldrin"]
    ["foo fighters","bar hoppers","baz aldrin"]
```

```
ghci 46> zipWith′ (*) (replicate 5 2) [1..]
    [2,4,6,8,10]
```

Note the repeated occurrence of *zipWith′* below:

```
ghci 47> zipWith′ (zipWith′ (*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
    [[3,4,6],[9,20,30],[10,12,12]]
```

As you can see, a single higher order function can be very versatile and can be used in very different ways.

3.3 Higher order functions structure complex computations

Imperative programming usually uses *for* loops, *while* loops, setting something to a variable, checking its state etc. to achieve some behavior, and then it wraps the entire sequence of statements in a 'function', which is not really a function in the mathematical sense but an 'execution unit' / a subroutine with a function-like interface. But this can miss important generalizations, hence the popularity of object-oriented programming etc.

In contrast, functional programming uses higher order functions directly (and these are really functions in the mathematical sense) to abstract away common patterns, like examining two lists in pairs and doing something with those pairs, or getting a set of solutions and eliminating the ones you don't need.

Purely functional languages like Haskell really push us to think of structuring programs in this way. For formal semanticists, this is very congenial because we analyze complex semantic phenomena in natural languages by following a similar strategy. We structure / decompose complex meanings, e.g., the meaning of a quantificational sentence like *No man left early*, by identifying various meaningful pieces, e.g., the restrictor property *man*, the nuclear scope property *left early*, the higher order relation between properties contributed by the determiner *no*, and we derive the meanings of more complex expressions, e.g., the noun phrase *no man* or the entire sentence *No man left early*, by putting these pieces together.

The fundamental insight is the same: higher order functions are an essential ingredient in structuring – and thereby understanding – complex computations, whether these computations are programs run by a computer or meanings grasped by a human mind.

3.4 A final example: implementing *flip*, a.k.a. passivization in English

We'll implement one more function that's already in the standard library, called *flip*. *flip* takes a binary function and returns a binary function that is like our original one except its two arguments are flipped.

This is very much like passivization in English, e.g., *The dwarf defeated the giant* gets 'flipped' / passivized as *The giant was defeated by the dwarf*.

```
ghci 48> let {flip' :: (a → b → c) → (b → a → c);
          flip' f = g
            where g x y = f y x}
```

The type declaration of *flip′* says that it takes a function with arguments of type a and b (in that order) and returns a function with arguments of type b and a (in that order).

Given that functions are curried by default and $\rightarrow$ is right-associative, the second pair of parentheses in the type signature $(a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c)$ is really unnecessary. But the main point of the *flip′* function can be more easily read from this type signature than from the equivalent and non-redundant one $(a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$.

Here are two example uses:

```
ghci 49> flip' zip [1,2,3,4,5] "hello"
    [('h',1),('e',2),('l',3),('l',4),('o',5)]
```

```
ghci 50> zipWith (flip' div) [2,2..] [10,8,6,4,2]
    [5,4,3,2,1]
```

Let's take a closer look at the **where** binding in the definition of *flip′*. We wrote that $g\ x\ y = f\ y\ x$, but an equivalent way of saying that is $g\ y\ x = f\ x\ y$. And since g is really just the name we use for *flip f′*, we can define this function more simply as:

```
ghci 51> let {flip'' :: (a → b → c) → (b → a → c);
              flip'' f y x = f x y}
```

Here, we take advantage of the fact that functions are curried: *flip'' f* is itself a function and this function is just like *f* except that when we call *f* with two arguments *x* and *y*, we need to call *flip'' f* with these arguments reversed to get the same result.

The function works just as before:

```
ghci 52> flip'' zip [1,2,3,4,5] "hello"
    [('h',1),('e',2),('l',3),('l',4),('o',5)]
```

```
ghci 53> zipWith (flip'' div) [2,2..] [10,8,6,4,2]
    [5,4,3,2,1]
```