Haskell: Lambda Expressions

Volker Sorge

March 20, 2012

λ-expressions (λ is the small Greek letter lambda) are a convenient way to easily create *anonymous functions* — functions that are not named and can therefore not be called out of context — that can be passed as parameters to higher order functions like `map`, `zip` etc. In most functional programming languages the syntax and usage of anonymous functions models itself after the λ-calculus, which was created by Alonzo Church in 1940. Church's aim was to create an unambiguous notation for functions and in particular to clearly identify the variables of a function.

A short λ-calculus Primer for Haskell

Fortunately, Haskell's syntax very closely follows the λ-calculus already, and we can restrict ourselves to a short primer.

The main idea of functional notation in the λ-calculus is to write all functions in prefix notation, to use currying to model function application to multiple parameters as iterated application to single parameters, and to designate parameters that can be instantiated during function application.

Prefix Notation

For example, applying a function f to a parameter x in λ-calculus is written as

$$fx$$

This is similar to Haskell, where for instance the application of the `even` predicate to the number `7` is written as `even 7`. In λ-calculus all functions are in prefix notation. In fact, this can be achieved in Haskell too. Consider the expression `3 + 4`. This can be written as `(+) 3 4` in Haskell, making it essentially equivalent to a function application of the form $f\,x\,y$ in λ-calculus.

Currying

Currying has been briefly discussed in the context of the Haskell functions `curry` and `uncurry`. The basic idea is that function application is only expressed in terms of applying a single function to a single argument. For example, the expression $f\,x\,y$ is a function application of f to two arguments x and y. Using currying we can write it as successive applications of the form $(fx)y$; that is, f is applied to x and the resulting function is applied to y.

In Haskell we can do the same: Take `(+) 3 4`. This can be written as `((+) 3) 4`, leading to the same result. To convince ourselves that the first part alone is indeed a function that can be applied to one argument, type `:t (+) 3` in the interpreter, which will yield a functional type. Likewise you can bind the function to some name by typing for example `let plusThree = ((+) 3)` and use it, e.g. `map plusThree [2,4,6]` yields the result `[5,7,9]`. In fact Haskell allows to be even briefer by simply writing `(+ 3)` for the function and for example write `map (+ 3) [2,4,6]`.

Code given between solid horizontal bars can be loaded into Haskell from a `.hs` file. Code between dotted horizontal bars can be typed in the interpreter at the prompt. It will always compute correctly but the result might occassionally cause a display error (e.g. if a particular type does not inherit from the `Show` type class).

λ Notation

Finally, λ-calculus allows one to clearly designate in every expression the variables that can be instantiated, by using a λ operator (which gave the calculus its name, in the first place). For example, if we want to express that in fx, the x is a parameter that can be instantiated, we write $\lambda x.fx$. Applying this function say to 3 by $(\lambda x.fx)3$ yields $f3$.

Likewise we can write $\lambda x \lambda y.f\,x\,y$, or for short $\lambda x\,y.f\,x\,y$, if we have two variable parameters x, y. Applying $(\lambda x\,y.f\,x\,y)\,3\,4$ yields $f\,3\,4$. Note, that the order of the variables in the so called λ *binder* matters as for example $(\lambda y\,x.f\,x\,y)\,3\,4$ yields $f\,4\,3$.

λ-calculus also can deal with *higher order functions*, i.e., functions that take other functions as input or return functions as result. For example, applying $(\lambda x\,y.f\,x\,y)3$ will return the function $\lambda y.f\,3\,y$. Likewise we can turn f into a variable by simply writing $(\lambda f\,x\,y.f\,x\,y)$.

In Haskell λ expressions are build very similarly, with very little notational difference: Instead of 'λ' we write the backslash '\' and the dot '.' is replaced by '$->$'. For example, we can write the final λ expression above as `\f x y -> f x y`. And applying it we can write in the interpreter

```
(\f x y -> f x y) (+) 3 4
```

which will result in 7. Similarly if we apply the λ function only to the arguments `(+)` and `3` we will get the `plusThree` function from above.

Usage

Anonymous functions are often used in a context where a functional expression is used only once in a higher order function. Although Haskell offers very sophisticated tools to assemble functional expressions without λ — more than many other functional programming languages — they do not always suffice and sometimes make code less easy to comprehend. For example the λ function

```
\x -> (4 * x + 3) `div` 3
```

can also be written with function composition as

```
(`div` 3) . (+ 3) . (* 4)
```

It is certainly in the eye of the beholder which one is easier to read or "more natural". We have to able to transform one into the other, however. Here is another example. Two implementations of a function that decides whether a given n is divisible by a list of *integers*.

```
divisible :: [Int] -> Int -> Bool
divisible divisors n = any (\d -> (mod n d)==0 ) divisors
```

```
divisible2 :: [Int] -> Int -> Bool
divisible2 divisors n = any ((== 0) . (mod n)) divisors
```

Haskell's syntactic sugar particularly breaks down in case a function uses multiple arguments multiple times. Consider the following function

```
\x y -> (x * x) + (y * y)
```

What will happen if we apply this function to a list of integers using `map`? For example:

```
map (\x y -> (x * x) + (y * y) ) [2,3,4]
```

As we have seen earlier in our discussion on currying, only one argument of the function will be bound in each application and will get a list of three functions:

```
[\y -> (2 * 2) + (y * y), \y -> (3 * 3) + (y * y), \y -> (4 * 4) + (y * y)]
```

To convince ourselves that this is indeed true, let's attempt the application of the first

```
(head $ map (\x y -> (x * x) + (y * y) ) [2,3,4]) 5
```

which yields 29.

More Higher Order

The previous example was a higher order function application in the sense that the higher order function map took a function as argument and returned a list of specialisations of this function. We also can take the other direction and generalise the function, by introducing more λ bound variables. For example,

```
\f g x y -> (x `f` x) `g` (y `f` y)
```

Observe, that the notation (x `f` x) allows us to use f in infix notation instead of prefix. Thus the expression is the same as \f g x y -> g (f x x) (f y y). Now

```
(\f g x y -> (x `f` x) `g` (y `f` y)) (*) (+) 2 5
```

results again in 29, whereas

```
(\f g x y -> (x `f` x) `g` (y `f` y)) (+) (*) 2 5
```

yields 40.

We can also use λ functions as arguments for other λ funtions.

```
g = \x -> x * x
h = \y -> g (g y)
j = h.h
```

While g is easy to identify as the square function, it is less straight forward to see what h and j exactly compute. Let's observe the function application step-by-step using λ notation:

$$g(g(y)) = (\lambda x.x * x)((\lambda x.x * x)y) = (\lambda x.x * x)(y * y) = ((y * y) * (y * y))$$

Thus h computes n^4. We now apply h to itself and get

$$\begin{aligned}(\lambda y.g(g(y)))(\lambda y.g(g(y))) &= (\lambda y.(y * y) * (y * y))(\lambda y.(y * y) * (y * y)) \\ &= \lambda y.((y * y) * (y * y)) * ((y * y) * (y * y)) * ((y * y) * (y * y)) * ((y * y) * (y * y))\end{aligned}$$

or in other words j computes n^{16}.

Finally, we can also generalise over ZF-expressions. For example, the following function

```
filtermap = \f g l -> [f x | x <-l , g x]
```

allows us to map f over a list l that has been filtered by criteria g. For example,

```
filtermap (* 3) even [1..10]}
```

triples all even numbers between 1 and 10.