Informatics 1

Functional Programming Lecture 8

Tuesday 23 October 2012

Lambda expressions, functions and binding

Don Sannella

University of Edinburgh

Required text and reading

Haskell: The Craft of Functional Programming (Third Edition),
Simon Thompson, Addison-Wesley, 2011.

Reading assignment

Monday 24 September 2012	Chapters 1–3 (pp. 1–66)
Monday 1 October 2012	Chapters 4–7 (pp. 67–176)
Monday 8 October 2012	Chapters 8–9 (pp. 177–212)
Monday 15 October 2012	Chapters 10–12 (pp. 213–286)
Monday 22 October 2012	*Class test*
Monday 29 October 2012	Chapters 13–14 (pp. 287–356)
Monday 5 November 2012	Chapters 15–16 (pp. 357–414)
Monday 12 November 2012	Chapters 17–21 (pp. 415–534)

Part I

Lambda expressions

A failed attempt to simplify

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map sqr (filter pos xs))
  where
  sqr x  =  x * x
  pos x  =  x > 0
```

The above *cannot* be simplified to the following:

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map (x * x) (filter (x > 0) xs))
```

A successful attempt to simplify

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map sqr (filter pos xs))
  where
  sqr x  =  x * x
  pos x  =  x > 0
```

The above *can* be simplified to the following:

```
f :: [Int] -> Int
f xs  =  foldr (+) 0
           (map (\x -> x * x)
              (filter (\x -> x > 0) xs))
```

Lambda calculus

```
f :: [Int] -> Int
f xs  =  foldr (+) 0
           (map (\x -> x * x)
              (filter (\x -> x > 0) xs))
```

The character `\` stands for λ, the Greek letter *lambda*.

Logicians write

`\x -> x > 0` as $\lambda x.\, x > 0$

`\x -> x * x` as $\lambda x.\, x \times x$.

Lambda calculus is due to the logician *Alonzo Church* (1903–1995).

Evaluating lambda expressions

```
    (\x -> x > 0) 3
=
    let x = 3 in x > 0
=
    3 > 0
=
    True

   (\x -> x * x) 3
=
   let x = 3 in x * x
=
   3 * 3
=
   9
```

Lambda expressions and currying

```
   (\x -> \y -> x + y) 3 4
=
   ((\x -> (\y -> x + y)) 3) 4
=
   (let x = 3 in \y -> x + y) 4
=
   (\y -> 3 + y) 4
=
   let y = 4 in 3 + y
=
   3 + 4
=
   7
```

Evaluating lambda expressions

The general rule for evaluating lambda expressions is

$$(\lambda x.\, N)\, M$$

$$=$$

$$(\texttt{let}\ x = M\ \texttt{in}\ N)$$

This is sometimes called the β rule (or beta rule).

Part II

Sections

Sections

`(> 0)` is shortand for `(\x -> x > 0)`

`(2 *)` is shortand for `(\x -> 2 * x)`

`(+ 1)` is shortand for `(\x -> x + 1)`

`(2 ^)` is shortand for `(\x -> 2 ^ x)`

`(^ 2)` is shortand for `(\x -> x ^ 2)`

Sections

```
f :: [Int] -> Int
f xs  =  foldr (+) 0
           (map (\x -> x * x)
             (filter (\x -> x > 0) xs))

f :: [Int] -> Int
f xs  =  foldr (+) 0 (map (^ 2) (filter (> 0) xs))
```

Part III

Composition

Composition

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x  =  f (g x)
```

Evaluation composition

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x  =  f (g x)

sqr :: Int -> Int
sqr x  =  x * x

pos :: Int -> Bool
pos x  =  x > 0

  (pos . sqr) 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

Compare and contrast

```
possqr :: Int -> Bool
possqr x  =  pos (sqr x)
```

```
  possqr 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

```
possqr :: Int -> Bool
possqr =  pos . sqr
```

```
  possqr 3
=
  (pos . sqr) 3
=
  pos (sqr 3)
=
  pos 9
=
  True
```

Composition is associative

```
(f . g) . h = f . (g . h)
```

```
    ((f . g) . h) x
=
    (f . g) (h x)
=
    f (g (h x))
=
    f ((g . h) x)
=
    (f . (g . h)) x
```

Thinking functionally

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map (^ 2) (filter (> 0) xs))

f :: [Int] -> Int
f  =  foldr (+) 0 . map (^ 2) . filter (> 0)
```

Applying the function

```
f :: [Int] -> Int
f  =  foldr (+) 0 . map (^ 2) . filter (> 0)

   f [1, -2, 3]
=
   (foldr (+) 0 . map (^ 2) . filter (> 0)) [1, -2, 3]
=
   foldr (+) 0 (map (^ 2) (filter (> 0) [1, -2, 3]))
=
   foldr (+) 0 (map (^ 2) [1, 3])
=
   foldr (+) 0 [1, 9]
=
   10
```

Part IV

Variables and binding

Variables

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

Variables—binding

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables—binding

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables—binding

```
x = 2
y = x+1
z = x+y*y

*Main> z
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables—renaming

```
xavier = 2
yolanda = xavier+1
zeuss = xavier+yolanda*yolanda

*Main> zeuss
11
```

Part V

Functions and binding

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

There are two *unrelated* uses of `x`!

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions—binding

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions—formal and actual parameters

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Functions—formal and actual parameters

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Functions—formal and actual parameters

```
f x = g x (x+1)
g x y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Functions—renaming

```
fred xavier = george xavier (xavier+1)
george xerox yolanda = xerox+yolanda*yolanda

*Main> fred 2
11
```

Different uses of `x` renamed to `xavier` and `xerox`.

Part VI

Variables in a where clause and binding

Variables in a where clause

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Variables in a where clause—binding

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables in a where clause—binding

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables in a where clause—binding

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables in a where clause—binding

```
f x = z
    where
    y = x+1
    z = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variables in a where clause—hole in scope

```
f x = z
    where
    y = x+1
    z = x+y*y

y = 5

*Main> y
5
```

Binding occurrence

Bound occurrence

Scope of binding

Part VII

Functions in a where clause and binding

Functions in a where clause

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Functions in a where clause—binding

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Variable `x` is still in scope within `g`!

Functions in a where clause—binding

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—binding

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—binding

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—hole in scope

```
f x = g (x+1)
    where
    g y = x+y*y

g z = z*z*z

*Main> g 2
8
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—pathological case

```
f x = f (x+1)
    where
    f y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—pathological case

```
f x = f (x+1)
    where
    f y = x+y*y

*Main> f 2
11
```

Binding occurrence

Bound occurrence

Scope of binding

Functions in a where clause—formals and actuals

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Functions in a where clause—formals and actuals

```
f x = g (x+1)
    where
    g y = x+y*y

*Main> f 2
11
```

Formal parameter

Actual parameter

Part VIII

Lambda expressions and binding

A wrong attempt to simplify

```
f :: [Int] -> [Int]
f xs  =  map (x * x) (filter (x > 0) xs)
```

This makes no sense—no binding occurrence of variable!

Lambda expressions

```
f :: [Int] -> [Int]
f xs  =
   map (\x -> x * x) (filter (\x -> x > 0) xs)
```

The character `\` stands for λ, the Greek letter *lambda*.

Logicians write

`(\x -> x * x)` as $(\lambda x.\, x \times x)$

`(\x -> x > 0)` as $(\lambda x.\, x > 0)$

Lambda expressions—binding

```
f :: [Int] -> [Int]
f xs  =  map (\x -> x*x) (filter (\x -> x > 0) xs)
```

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—binding

```
f :: [Int] -> [Int]
f xs  =  map (\x -> x*x) (filter (\x -> x > 0) xs)
```

Binding occurrence

Bound occurrence

Scope of binding

Part IX

Lambda expressions explain binding

Lambda expressions explain binding

A variable binding can be rewritten using a lambda expression and an application:

$$
\begin{aligned}
& (N \texttt{ where } x = M) \\
= & \\
& (\lambda x.\, N)\, M \\
= & \\
& (\texttt{let } x = M \texttt{ in } N)
\end{aligned}
$$

A function binding can be written using an application on the left or a lambda expression on the right:

$$
\begin{aligned}
& (M \texttt{ where } f\ x = N) \\
= & \\
& (M \texttt{ where } f = \lambda x.\, N)
\end{aligned}
$$

Lambda expressions and binding constructs

```
      f 2
      where
      f x  =  x+y*y
           where
           y = x+1
=
      f 2
      where
      f  =  \x -> (x+y*y where y = x+1)
=
      f 2
      where
      f  =  \x -> ((\y -> x+y*y) (x+1))
=
      (\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Evaluating lambda expressions

```
    (\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
=
    (\x -> ((\y -> x+y*y) (x+1))) 2
=
    (\y -> 2+y*y) (2+1)
=
    (\y -> 2+y*y) 3
=
    2+3*3
=
    11
```

Part X

Additional material:

Lambda expressions and binding, revisited

Lambda expressions—binding

```
(\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—binding

```
(\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—binding

```
(\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Binding occurrence

Bound occurrence

Scope of binding

Lambda expressions—formals and actuals

```
(\f -> f 2) (\x -> ((\y -> x+y*y) (x+1)))
```

Formal parameter

Actual parameter

Lambda expressions—formals and actuals

```
(\x -> ((\y -> x+y*y) (x+1))) 2
```

Formal parameter

Actual parameter

Lambda expressions—formals and actuals

```
(\y -> 2+y*y) (2+1)
```

Formal parameter

Actual parameter

Part XI

Additional material:
Comprehensions and binding

Comprehensions

```
f :: [Int] -> [Int]
f xs  =  [ x*x | x <- xs, x > 0 ]

*Main> f [1,-2,3]
[1,9]
```

Comprehensions—binding

```
f :: [Int] -> [Int]
f xs  =  [ x*x | x <- xs, x > 0 ]

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

Comprehensions—binding

```
f :: [Int] -> [Int]
f xs  =  [ x*x | x <- xs, x > 0 ]

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

Comprehensions—pathological case

```
f :: [Int] -> [Int]
f x  =  [ x*x | x <- x, x > 0 ]

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding – Note hole in scope!

Squares of Positives—pathological case

```
f :: [Int] -> [Int]
f x  =  [ x*x | x <- x, x > 0 ]

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

List comprehension with two qualifiers

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

List comprehension with two qualifiers—binding

```
f n = [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

List comprehension with two qualifiers—binding

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

List comprehension with two qualifiers—binding

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

Part XII

Additional material:
Higher-order functions and binding

Higher-order functions

```
f :: [Int] -> [Int]
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Higher order functions—binding

```
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence—not shown (in standard prelude)

Bound occurrence

Scope of binding

Higher-order functions—binding

```
f xs  =  map sqr (filter pos xs)
  where
  sqr x  =  x*x
  pos x  =  x > 0

*Main> f [1,-2,3]
[1,9]
```

Binding occurrence—not shown (in standard prelude)

Bound occurrence

Scope of binding