Informatics 1

Functional Programming Lecture 12

Tuesday 4 November 2008

Binding and lambda calculus

Philip Wadler

University of Edinburgh

Required reading

Haskell: The Craft of Functional Programming, Second Edition,
Simon Thompson, Addison-Wesley, 1999.

Thompson, Chapters 1–3 (pp. 1–52): by Mon 29 Sep 2008.
Thompson, Chapters 4–5 (pp. 53–95): by Mon 6 Oct 2008.
Thompson, Chapters 6–7 (pp. 96–134): by Mon 13 Oct 2008.
Thompson, Chapters 8–9 (pp. 135–166): by Mon 20 Oct 2008.
Thompson, Chapters 10–11 (pp. 167–209): by Mon 3 Nov 2008.
Thompson, Chapters 12–14 (pp. 210–241): by Mon 10 Nov 2006.

Thompson and other books available in ITO.

Part I

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 II

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 III

Variables in a where clause

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 IV

Functions in a where clause

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 V

Squares of Positives

Squares of Positives—comprehension

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

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

Squares of Positives—binding

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

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—binding

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

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—pathological case

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

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

Binding occurrence

Bound occurrence

Scope of binding – Note hole in scope!

Squares of Positives—pathological case

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

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—higher-order functions

```
squarePositives :: [Int] -> [Int]
squarePositives xs   =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Squares of Positives—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x      =  x*x
  positive x    =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—binding

```
squarePositives xs   =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence—not shown (in standard prelude)

Bound occurrence

Scope of binding

Squares of Positives—binding

```
squarePositives xs  =  map square (filter positive xs)
  where
  square x     =  x*x
  positive x   =  x > 0

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

Binding occurrence—not shown (in standard prelude)

Bound occurrence

Scope of binding

Part VI

Lambda expressions

Squares of Positives—a wrong attempt to simplify

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

This makes no sense—no binding occurrence of variable!

Squares of Positives—lambda expressions

```
squarePositives :: [Int] -> [Int]
squarePositives 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 $(\lambda x.x \times x)$

Squares of Positives—lambda expressions

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

Binding occurrence

Bound occurrence

Scope of binding

Squares of Positives—lambda expressions

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

Binding occurrence

Bound occurrence

Scope of binding

Evaluating lambda expressions

```
    map (\x -> x*x) [1,2,3]
=
    [(\x -> x*x) 1, (\x -> x*x) 2, (\x -> x*x) 3]
=
    [1*1, 2*2, 3*3]
=
    [1, 4, 9]
```

The general rule

To apply a function to an argument, substitute the argument for the bound variable:

$$
\begin{aligned}
& (\lambda x.\, N)\; M \\
= \quad & N[M/x]
\end{aligned}
$$

Here $N[M/x]$ is the result of substituting term M for each occurrence of variable x in term N.

For example, if x is `y`, and N is `y*y` and M is `2`:

```
  (\y -> y*y) 2
= 2*2
```

Lambda expressions and binding constructs

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

$$\begin{aligned} & (N \;\texttt{where}\; x = M) \\ = \quad & (\lambda x.\, N)\; M \\ = \quad & N[M/x] \end{aligned}$$

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

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

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
```

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 VII

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)`

Squares of Positives—sections

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

Part VIII

List comprehensions

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

Evaluating a list comprehension

```
    [ (i,j) | i <- [1..3], j <- [i..3] ]
=
    [ (1,j) | j <- [1..3] ] ++
    [ (2,j) | j <- [2..3] ] ++
    [ (3,j) | j <- [3..3] ]
=
    [(1,1),(1,2),(1,3)] ++
    [(2,2),(2,3)] ++
    [(3,3)]
=
    [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Another example

```
    [ (i,j) | i <- [1..3], j <- [1..3], i <= j ]
=
    [ (1,j) | j <- [1..3], 1 <= j ] ++
    [ (2,j) | j <- [1..3], 2 <= j ] ++
    [ (3,j) | j <- [1..3], 3 <= j ]
=
    [(1,1)|1<=1] ++ [(1,2)|1<=2] ++ [(1,3)|1<=3] ++
    [(2,1)|2<=1] ++ [(2,2)|2<=2] ++ [(2,3)|2<=3] ++
    [(3,1)|3<=1] ++ [(3,2)|3<=2] ++ [(3,3)|3<=3]
=
    [(1,1)] ++ [(1,2)] ++ [(1,3)] ++
    []      ++ [(2,2)] ++ [(2,3)] ++
    []      ++ []      ++ [(3,3)]
=
    [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Defining list comprehensions

$$
\begin{aligned}
[\, e \mid x \leftarrow l,\ q \,] &= \texttt{concat}\ (\texttt{map}\ (\lambda x.\, [\, e \mid q \,])\ xs) \\
&= l \ggg\!= \lambda x.\, [\, e \mid q \,] \\
[\, e \mid p,\ q \,] &= \texttt{if}\ p\ \texttt{then}\ [:\ e \mid q\,]\ \texttt{else}\ [\,] \\
&= \texttt{guard}\ p \gg [\, e \mid q \,] \\
[\, e \mid \bullet \,] &= [\, e \,]
\end{aligned}
$$

$$
\begin{aligned}
xs \ggg\!= f &= \texttt{concat}\ (\texttt{map}\ f\ xs) \\
xs \gg ys &= \texttt{concat}\ (\texttt{map}\ (\lambda x.\, ys)\ xs) \\
\texttt{guard}\ p &= \texttt{if}\ p\ \texttt{then}\ [()]\ \texttt{else}\ [\,]
\end{aligned}
$$

Examples

```
    [ x*x | x <- xs ]
=   xs >>= \x ->
    [ x*x ]

    [ x*x | x <- xs, x > 0 ]
=   xs >>= \x ->
    guard (x > 0) >>
    [ x*x ]

    [ (i,j) | i <- [1..3], j <- [i..3] ]
=   [1..3] >>= \i ->
    [i..3] >>= \j ->
    [ (i,j) ]

    [ (i,j) | i <- [1..3], j <- [1..3], i <= j ]
=   [1..3] >>= \i ->
    [1..3] >>= \j ->
    guard (i <= j) >>
    [ (i,j) ]
```