

Informatics 1A
Functional Programming Lecture 8

Lambda expressions

Ohad Kammar
The University of Edinburgh

Part I

Currying

How to add two numbers

```
add :: Int -> Int -> Int  
add x y = x + y
```

```
    add 3 4  
=  
    3 + 4  
=  
    7
```

How to add two numbers

```
add :: Int -> (Int -> Int)
(add x) y = x + y
```

```
(add 3) 4
=
3 + 4
=
7
```

A function of two numbers
is the same as
a function of the first number that returns
a function of the second number.

Currying

```
add :: Int -> (Int -> Int)
```

```
add x = g
```

```
  where
```

```
    g :: Int -> Int
```

```
    g y = (x + y)
```

```
(add 3) 4
```

```
=
```

```
g 4 where g y = 3 + y
```

```
=
```

```
3 + 4
```

```
=
```

```
7
```

This idea is named for *Haskell Curry* (1900–1982).

It also appears in the work of *Moses Schönfinkel* (1889–1942),
and *Gottlob Frege* (1848–1925).

Partial evaluation

```
foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f u []          = u
foldr f u (x:xs)      = f x (foldr f u xs)
```

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

is equivalent to

```
foldr :: (a -> a -> a) -> a -> ([a] -> a)
foldr f u []          = u
foldr f u (x:xs)      = f x (foldr f u xs)
```

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

Sum, Product, Concat, And, Or

```
sum      :: [Int] -> Int
sum xs   = foldr (+) 0 xs
```

```
product  :: [Int] -> Int
product xs = foldr (*) 1 xs
```

```
concat   :: [[a]] -> [a]
concat xs = foldr (++) [] xs
```

```
and       :: [Bool] -> Bool
and xs    = foldr (&&) True xs
```

```
or        :: [Bool] -> Bool
or xs     = foldr (||) False xs
```

Sum, Product, Concat, And, Or: simplified

```
sum      :: [Int] -> Int
sum      = foldr (+) 0
```

```
product  :: [Int] -> Int
product  = foldr (*) 1
```

```
concat   :: [[a]] -> [a]
concat   = foldr (++) []
```

```
and       :: [Bool] -> Bool
and       = foldr (&&) True
```

```
or        :: [Bool] -> Bool
or        = foldr (||) False
```

Part II

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

$$\begin{aligned} & (\lambda x. x > 0) \ 3 \\ = & \quad 3 > 0 \\ = & \quad \text{True} \end{aligned}$$
$$\begin{aligned} & (\lambda x. x * x) \ 3 \\ = & \quad 3 * 3 \\ = & \quad 9 \end{aligned}$$

Lambda expressions and currying

$$\begin{aligned} & (\lambda x. \lambda y. x + y) \ 3 \ 4 \\ = & \\ & ((\lambda x. \lambda y. x + y) \ 3) \ 4 \\ = & \\ & (\lambda y. 3 + y) \ 4 \\ = & \\ & 3 + 4 \\ = & \\ & 7 \end{aligned}$$

The beta rule

The general rule for evaluating lambda expressions is called the β rule, after the Greek letter beta:

$$(\lambda x. N) M = N[x := M]$$

Here N and M are arbitrary expressions, and $N[x := M]$ is N with each free occurrence of x replaced by M .

$$\begin{aligned} & (\lambda x y. x + y) 3 4 \\ = & ((\lambda x. (\lambda y. x + y)) 3) 4 \\ = & ((\lambda y. x + y)[x := 3]) 4 \\ = & (\lambda y. 3 + y) 4 \\ = & (3 + y)[y := 4] \\ = & 3 + 4 \\ = & 7 \end{aligned}$$

Part III

Sections

Sections

(> 0) *stands for* $(\backslash x \rightarrow x > 0)$

$(2 \ast)$ *stands for* $(\backslash x \rightarrow 2 \ast x)$

$(+ 1)$ *stands for* $(\backslash x \rightarrow x + 1)$

$(2 \wedge)$ *stands for* $(\backslash x \rightarrow 2 \wedge x)$

$(\wedge 2)$ *stands for* $(\backslash x \rightarrow x \wedge 2)$

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 IV

Composition

Composition

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
 $(f . g) x = f (g x)$

Evaluating 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

$$(f \cdot g) \cdot h = f \cdot (g \cdot h)$$

$$\begin{aligned} & ((f \cdot g) \cdot h) x \\ = & (f \cdot g) (h x) \\ = & f (g (h x)) \\ = & f ((g \cdot h) x) \\ = & (f \cdot (g \cdot h)) x \end{aligned}$$

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
```