Informatics 1
Functional Programming Lecture 8

Lambda expressions, functions and binding

Don Sannella
University of Edinburgh
Part I

Lambda expressions
A failed attempt to simplify

\[
f :: [\text{Int}] \rightarrow \text{Int} \\
f \, \text{xs} = \text{foldr} (+) 0 (\text{map} \, \text{sqr} \, (\text{filter} \, \text{pos} \, \text{xs})) \\
\quad \text{where} \\
\quad \text{sqr} \, x = x \times x \\
\quad \text{pos} \, x = x > 0
\]

The above \textit{cannot} be simplified to the following:

\[
f :: [\text{Int}] \rightarrow \text{Int} \\
f \, \text{xs} = \text{foldr} (+) 0 (\text{map} \, (x \times x) \, (\text{filter} \, (x > 0) \, \text{xs}))
\]

Looking at the previous example (sum of squares of positive numbers): can't I just write the bodies of \text{sqr} and \text{pos} "in place"?

Computer says: "\text{x is not in scope}"
That is, it doesn't know what you mean by \text{x}.

We need some way of saying: assuming that \text{x} is the argument, return \text{x} \times \text{x}
A successful attempt to simplify

\[
f :: [\text{Int}] \to \text{Int}
\]
\[
f \, \text{xs} = \text{foldr} \ (+) \ 0 \ (\text{map} \ \text{sqr} \ (\text{filter} \ \text{pos} \ \text{xs}))
\]
where
\[
\text{sqr} \, x = x * x
\]
\[
\text{pos} \, x = x > 0
\]

The above \textit{can} be simplified to the following:

\[
f :: [\text{Int}] \to \text{Int}
\]
\[
f \, \text{xs} = \text{foldr} \ (+) \ 0
\]
\[
(\text{map} \ (\lambda x \to x * x)
\]
\[
(\text{filter} \ (\lambda x \to x > 0) \ \text{xs})
\]

\(\lambda x \to x * x\) means: asassuming that \(x\) is the argument, return \(x * x\)
\(x\) is arbitrary - you could using any identifier, and it could be different for the two functions.
Lambda calculus

\[ f :: [\text{Int}] \rightarrow \text{Int} \]
\[ f \text{ xs} = \text{foldr} \ (+) \ 0 \]
\[ (\text{map} (\lambda x \rightarrow x \times x) \]
\[ (\text{filter} (\lambda x \rightarrow x > 0) \text{ xs})) \]

The character \ stands for \( \lambda \), the Greek letter \textit{lambda}. Logicians write
\[ \lambda x \rightarrow x > 0 \] as \( \lambda x. x > 0 \)
\[ \lambda x \rightarrow x \times x \] as \( \lambda x. x \times x \).

Lambda calculus is due to the logician \textit{Alonzo Church} (1903–1995).

\( \backslash \) is the closest thing on the keyboard to lambda.
The lambda calculus is a theory of functions, that was designed before computers existed.

Lambda expressions finally came to Java in 2014, only about 55 years after they came to functional programming.
Evaluating lambda expressions

\( (\lambda x \rightarrow x > 0) \ 3 \)

= 

\( \text{let } x = 3 \text{ in } x > 0 \)

= 

\( 3 > 0 \)

= 

True

\( (\lambda x \rightarrow x * x) \ 3 \)

= 

\( \text{let } x = 3 \text{ in } x * x \)

= 

\( 3 * 3 \)

= 

9

This is how you evaluate lambda-expressions. It's just a function, so \( (\lambda x \rightarrow x > 0) \ 3 \) is \( 3 > 0 \)

We can do that in 2 steps, by using

\( \text{let } x = 3 \text{ in } ... \)

to express the passing of the argument to the function body.
Lambda expressions and currying

$(\lambda x \to \lambda y \to x + y) \ 3 \ 4$

$= (\lambda x \to (\lambda y \to x + y)) \ 3 \ 4$

$= (\text{let } x = 3 \ \text{in } \lambda y \to x + y) \ 4$

$= (\lambda y \to 3 + y) \ 4$

$= \text{let } y = 4 \ \text{in } 3 + y$

$= 3 + 4$

$= 7$

We can use this notation to express directly what currying is doing. $(\lambda y \to 3 + y)$ is the function that is returned from $(\lambda x \to \lambda y \to x + y)$ when it is applied to 3.
Evaluating lambda expressions

The general rule for evaluating lambda expressions is

\[(\lambda x. \ N) \ M \]

\[=\]

\[(\text{let } x = M \text{ in } N)\]

If you have a lambda-expression applied to an argument ...

... replace x by M when evaluating N

This is sometimes called the $\beta$ rule (or beta rule).
Part II

Sections
Sections

(> 0) is shorthand for (\x -> x > 0)

(2 *) is shorthand for (\x -> 2 * x)

(+ 1) is shorthand for (\x -> x + 1)

(2 ^) is shorthand for (\x -> 2 ^ x)  exponentiation

(^ 2) is shorthand for (\x -> x ^ 2)  squaring

SECTIONS are a convenient shorthand for writing partially-applied functions.
A binary operator with an argument on the left or right, in parentheses.
Explained using lambda-expressions.
Where x goes depends on where the argument was - x goes in place of the missing argument.
It's the fact that these functions are curried that makes this work.
Sections

\[
f :: [\text{Int}] \rightarrow \text{Int}
f \text{ xs} = \text{foldr} \ (+) \ 0 \\
\quad (\text{map} \ (\lambda x \rightarrow x \times x) \\
\quad \quad (\text{filter} \ (\lambda x \rightarrow x > 0) \ \text{xs}))
\]

\[
f :: [\text{Int}] \rightarrow \text{Int}
f \text{ xs} = \text{foldr} \ (+) \ 0 \ (\text{map} \ (^{2}) \ (\text{filter} \ (> 0) \ \text{xs}))
\]

We can write the previous example really compactly using sections.
Part III

Composition
Composition

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

The composition operator is built in to Haskell.
Takes two functions and produces a function that does one and then the other.
Try to figure out why the type is as written above rather than
(a -> b) -> (b -> c) -> (a -> c)
Evaluating composition

\[(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)\]
\[(f \cdot g) \, x = f \, (g \, x)\]

\[sqr :: \text{Int} \rightarrow \text{Int}\]
\[sqr \, x = x \times x\]

\[pos :: \text{Int} \rightarrow \text{Bool}\]
\[pos \, x = x > 0\]

\[(pos \cdot sqr) \, 3\]
\[=\]
\[pos \, (sqr \, 3)\]
\[=\]
\[pos \, 9\]
\[=\]
\[True\]
Compare and contrast

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

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

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

possqr 3 =
  (pos . sqr) 3
  =
  pos (sqr 3)
  =
  pos 9
  =
  True
```
Composition is associative

\[(f \ . \ g) \ . \ h = f \ . \ (g \ . \ h)\]

\[\begin{align*}
((f \ . \ g) \ . \ h) \ x &= \\
(f \ . \ g) \ (h \ x) &= \\
f \ (g \ (h \ x)) &= \\
f \ ((g \ . \ h) \ x) &= \\
(f \ . \ (g \ . \ h)) \ x &=
\end{align*}\]
Thinking functionally

\[
f :: [\text{Int}] \rightarrow \text{Int} \\
f \; \text{xs} \; = \; \text{foldr} \; (+) \; 0 \; (\text{map} \; (^2) \; (\text{filter} \; (> \; 0) \; \text{xs}))
\]

\[
f :: [\text{Int}] \rightarrow \text{Int} \\
f \; = \; \text{foldr} \; (+) \; 0 \; . \; \text{map} \; (^2) \; . \; \text{filter} \; (> \; 0)
\]

So I don't need to think about the argument \text{xs} at all. I can write the earlier example using composition. I don't need parentheses because composition is associative - it doesn't matter which way they are added.
Applying the function

\[
f :: [\text{Int}] \rightarrow \text{Int}
f = \text{foldr} (+) 0 \ . \ \text{map} (\text{\^} 2) \ . \ \text{filter} (> 0)
\]

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

Here's how it works.
Part IV

Variables and binding
Variables

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

*Main>  z
11

Here's a very simple Haskell program.
If you're used to C or Java or Visual Basic, you might think that x, y, z are boxes that you put values in.
Later you might change the value in the box using something like x = x + 1.
Not in functional programming!
This is BINDING, not ASSIGNMENT.
We say "x is bound to 2". x is just a name for 2. x means 2 throughout the program.
Part V

Lambda expressions explain binding
Lambda expressions explain binding

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

\[(N \text{ where } x = M)\]
\[=\]
\[\left(\lambda x. N \right) M\]
\[=\]
\[(\text{let } x = M \text{ in } N)\]

Here is the key rule. In both cases, we are replacing x in N by M

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

\[(M \text{ where } f \ x = N)\]
\[=\]
\[(M \text{ where } f = \lambda x. N)\]

We can write function definitions using lambda, too. Everything in functional programming can be written using lambda.
Lambda expressions and binding constructs

\[ f \ 2 \]

where
\[ f \ x \ = \ x + y \ast y \]

where
\[ y \ = \ x + 1 \]

= 

\[ f \ 2 \]

where
\[ f \ = \ \lambda x . (x + y \ast y \text{ where } y = x + 1) \]

= 

\[ f \ 2 \]

where
\[ f \ = \ \lambda x . ((\lambda y . x + y \ast y) \ (x + 1)) \]

= 

\[ (\lambda f . f \ 2) \ (\lambda x . ((\lambda y . x + y \ast y) \ (x + 1))) \]

Here's an example, expanding the wheres and the function bindings. Everything turns into a big lambda-expression with application.
Evaluating lambda expressions

\((\lambda f \rightarrow f \ 2) \ (\lambda x \rightarrow ((\lambda y \rightarrow x + y \times y) \ (x + 1)))\)

= 
\((\lambda x \rightarrow ((\lambda y \rightarrow x + y \times y) \ (x + 1))) \ 2\)

= 
\((\lambda y \rightarrow x + y \times y) \ (x + 1)) \ 2\)

= 
\((\lambda y \rightarrow x + y \times y) \ (2 + 1)\)

= 
\((\lambda y \rightarrow x + y \times y) \ 3\)

= 
2 + 3 \times 3

= 
11

And we know how to evaluate lambda-expressions applied to argument: replace formal parameter by actual parameter.

Everything in functional programming can be explained by lambda expressions. You could view them as the assembly language of functional programming.

Even lower level: COMBINATORS. All of lambda calculus can be boiled down to S and K, defined like this:
K x y = x
S x y z = (x z) (y z)
Combinators are like the quarks of computing.