

Programming Language Concepts: Lecture 15

Madhavan Mukund

Chennai Mathematical Institute

`madhavan@cmi.ac.in`

`http://www.cmi.ac.in/~madhavan/courses/pl2009`

PLC 2009, Lecture 15, 18 March 2009

λ -calculus

- ▶ A notation for computable functions
 - ▶ Alonzo Church

λ -calculus

- ▶ A notation for computable functions
 - ▶ Alonzo Church
- ▶ How do we describe a function?
 - ▶ By its graph — a binary relation between **domain** and **codomain**
 - ▶ Single-valued
 - ▶ **Extensional** — graph completely defines the function

λ -calculus

- ▶ A notation for computable functions
 - ▶ Alonzo Church
- ▶ How do we describe a function?
 - ▶ By its graph — a binary relation between **domain** and **codomain**
 - ▶ Single-valued
 - ▶ **Extensional** — graph completely defines the function
- ▶ An extensional definition is not suitable for computation
 - ▶ All sorting functions are the same!

λ -calculus

- ▶ A notation for computable functions
 - ▶ Alonzo Church
- ▶ How do we describe a function?
 - ▶ By its graph — a binary relation between **domain** and **codomain**
 - ▶ Single-valued
 - ▶ **Extensional** — graph completely defines the function
- ▶ An extensional definition is not suitable for computation
 - ▶ All sorting functions are the same!
- ▶ Need an **intensional** definition
 - ▶ How are outputs computed from inputs?

λ -calculus: syntax

- ▶ Assume a set Var of variables
- ▶ Set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MM'$$

where $x \in Var$, $M, M' \in \Lambda$.

λ -calculus: syntax

- ▶ Assume a set Var of variables
- ▶ Set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MM'$$

where $x \in Var$, $M, M' \in \Lambda$.

- ▶ $\lambda x.M$: Abstraction
 - ▶ A function of x with computation rule M .
 - ▶ “Abstracts” the computation rule M over arbitrary input values x
 - ▶ Like writing $f(x) = e$ without assigning a name f

λ -calculus: syntax

- ▶ Assume a set Var of variables
- ▶ Set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MM'$$

where $x \in Var$, $M, M' \in \Lambda$.

- ▶ $\lambda x.M$: Abstraction
 - ▶ A function of x with computation rule M .
 - ▶ “Abstracts” the computation rule M over arbitrary input values x
 - ▶ Like writing $f(x) = e$ without assigning a name f
- ▶ MM' : Application
 - ▶ Apply the function M to the argument M'

λ -calculus: syntax ...

- ▶ Can write expressions such as xx — no types!

λ -calculus: syntax ...

- ▶ Can write expressions such as xx — no types!
- ▶ What can we do without types?

λ -calculus: syntax . . .

- ▶ Can write expressions such as xx — no types!
- ▶ What can we do without types?
 - ▶ Set theory as a basis for mathematics
 - ▶ Bit strings in memory

λ -calculus: syntax . . .

- ▶ Can write expressions such as xx — no types!
- ▶ What can we do without types?
 - ▶ Set theory as a basis for mathematics
 - ▶ Bit strings in memory
- ▶ In an untyped world, some data is *meaningful*

λ -calculus: syntax . . .

- ▶ Can write expressions such as xx — no types!
- ▶ What can we do without types?
 - ▶ Set theory as a basis for mathematics
 - ▶ Bit strings in memory
- ▶ In an untyped world, some data is **meaningful**
- ▶ Functions manipulate meaningful data to yield meaningful data

λ -calculus: syntax . . .

- ▶ Can write expressions such as xx — no types!
- ▶ What can we do without types?
 - ▶ Set theory as a basis for mathematics
 - ▶ Bit strings in memory
- ▶ In an untyped world, some data is **meaningful**
- ▶ Functions manipulate meaningful data to yield meaningful data
- ▶ Can also apply functions to non-meaningful data, but the result has no significance

The computation rule β

- ▶ Basic rule for computing (rewriting) is called β

$$(\lambda x.M)M' \rightarrow_{\beta} M\{x \leftarrow M'\}$$

- ▶ $M\{x \leftarrow M'\}$: substitute **free** occurrences of x in M by M'

The computation rule β

- ▶ Basic rule for computing (rewriting) is called β

$$(\lambda x.M)M' \rightarrow_{\beta} M\{x \leftarrow M'\}$$

- ▶ $M\{x \leftarrow M'\}$: substitute **free** occurrences of x in M by M'
- ▶ This is the normal rule we use for functions:

The computation rule β

- ▶ Basic rule for computing (rewriting) is called β

$$(\lambda x.M)M' \rightarrow_{\beta} M\{x \leftarrow M'\}$$

- ▶ $M\{x \leftarrow M'\}$: substitute **free** occurrences of x in M by M'
- ▶ This is the normal rule we use for functions:

$$f(x) = 2x^2 + 3x + 4$$

The computation rule β

- ▶ Basic rule for computing (rewriting) is called β

$$(\lambda x.M)M' \rightarrow_{\beta} M\{x \leftarrow M'\}$$

- ▶ $M\{x \leftarrow M'\}$: substitute **free** occurrences of x in M by M'
- ▶ This is the normal rule we use for functions:

$$f(x) = 2x^2 + 3x + 4$$

$$f(7) = 2 \cdot 7^2 + 3 \cdot 7 + 4 = (2x^2 + 3x + 4)\{x \leftarrow 7\}.$$

The computation rule β

- Basic rule for computing (rewriting) is called β

$$(\lambda x.M)M' \rightarrow_{\beta} M\{x \leftarrow M'\}$$

- $M\{x \leftarrow M'\}$: substitute **free** occurrences of x in M by M'
- This is the normal rule we use for functions:

$$f(x) = 2x^2 + 3x + 4$$

$$f(7) = 2 \cdot 7^2 + 3 \cdot 7 + 4 = (2x^2 + 3x + 4)\{x \leftarrow 7\}.$$

- β is the **only** rule we need!

The computation rule β

- ▶ Basic rule for computing (rewriting) is called β

$$(\lambda x.M)M' \rightarrow_{\beta} M\{x \leftarrow M'\}$$

- ▶ $M\{x \leftarrow M'\}$: substitute **free** occurrences of x in M by M'
- ▶ This is the normal rule we use for functions:

$$f(x) = 2x^2 + 3x + 4$$

$$f(7) = 2 \cdot 7^2 + 3 \cdot 7 + 4 = (2x^2 + 3x + 4)\{x \leftarrow 7\}.$$

- ▶ β is the **only** rule we need!
- ▶ MM' is meaningful only if M is of the form $\lambda x.M''$
 - ▶ Cannot do anything with expressions like xx

Variable capture

- ▶ Consider $(\lambda x. (\lambda y. xy))y$

Variable capture

- ▶ Consider $(\lambda x. (\lambda y. xy))y$
- ▶ β yields $\lambda y. yy$
 - ▶ The y substituted for inner x has been “confused” with the y bound by λy
- ▶ Rename bound variables to avoid capture

$$(\lambda x. (\lambda y. xy))y = (\lambda x. (\lambda z. xz))y \rightarrow_{\beta} \lambda z. yz$$

- ▶ Renaming bound variables does not change the function
 - ▶ $f(x) = 2x + 5$ vs $f(z) = 2z + 5$

Variable capture

Formally, bound and free variables are defined as

- ▶ $FV(x) = \{x\}$, for any variable x
- ▶ $FV(\lambda x.M) = FV(M) - \{x\}$
- ▶ $FV(MM') = FV(M) \cup FV(M')$

Variable capture

Formally, bound and free variables are defined as

- ▶ $FV(x) = \{x\}$, for any variable x
- ▶ $FV(\lambda x.M) = FV(M) - \{x\}$
- ▶ $FV(MM') = FV(M) \cup FV(M')$

- ▶ $BV(x) = \emptyset$, for any variable x
- ▶ $BV(\lambda x.M) = BV(M) \cup \{x\}$
- ▶ $BV(MM') = BV(M) \cup BV(M')$

Variable capture

Formally, bound and free variables are defined as

- ▶ $FV(x) = \{x\}$, for any variable x
- ▶ $FV(\lambda x.M) = FV(M) - \{x\}$
- ▶ $FV(MM') = FV(M) \cup FV(M')$

- ▶ $BV(x) = \emptyset$, for any variable x
- ▶ $BV(\lambda x.M) = BV(M) \cup \{x\}$
- ▶ $BV(MM') = BV(M) \cup BV(M')$

When we apply β to MM' , assume that we always rename the bound variables in M to avoid “capturing” free variables from M' .

Encoding arithmetic

In set theory, use nesting depth to encode numbers

- ▶ Encoding of n : $\langle n \rangle$
- ▶ $\langle n \rangle = \{\langle 0 \rangle, \langle 1 \rangle, \dots, \langle n-1 \rangle\}$

Encoding arithmetic

In set theory, use nesting depth to encode numbers

- ▶ Encoding of n : $\langle n \rangle$
- ▶ $\langle n \rangle = \{\langle 0 \rangle, \langle 1 \rangle, \dots, \langle n-1 \rangle\}$

Thus

$$\begin{aligned}0 &= \emptyset \\1 &= \{\emptyset\} \\2 &= \{\emptyset, \{\emptyset\}\} \\3 &= \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\&\dots\end{aligned}$$

Encoding arithmetic

In set theory, use nesting depth to encode numbers

- ▶ Encoding of n : $\langle n \rangle$
- ▶ $\langle n \rangle = \{\langle 0 \rangle, \langle 1 \rangle, \dots, \langle n-1 \rangle\}$

Thus

$$\begin{aligned} 0 &= \emptyset \\ 1 &= \{\emptyset\} \\ 2 &= \{\emptyset, \{\emptyset\}\} \\ 3 &= \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ &\dots \end{aligned}$$

In λ -calculus, encode n by number of times we apply a function

Encoding arithmetic ...

Church numerals

$$\begin{aligned}\langle 0 \rangle &= \lambda f x. x \\ \langle n + 1 \rangle &= \lambda f x. f(\langle n \rangle f x)\end{aligned}$$

Encoding arithmetic ...

Church numerals

$$\begin{aligned}\langle 0 \rangle &= \lambda f x. x \\ \langle n + 1 \rangle &= \lambda f x. f(\langle n \rangle f x)\end{aligned}$$

For instance

$$\langle 1 \rangle = \lambda f x. f(\langle 0 \rangle f x) = \lambda f x. (f((\lambda f x. x) f x))$$

Encoding arithmetic ...

Church numerals

$$\begin{aligned}\langle 0 \rangle &= \lambda f x. x \\ \langle n + 1 \rangle &= \lambda f x. f(\langle n \rangle f x)\end{aligned}$$

For instance

$$\langle 1 \rangle = \lambda f x. f(\langle 0 \rangle f x) = \lambda f x. (f((\lambda f x. x) f x))$$

Note that $\langle 0 \rangle g y \rightarrow_{\beta} (\lambda x. x) y \rightarrow_{\beta} y$.

Hence

$$\langle 1 \rangle = \dots = \lambda f x. \underbrace{(f((\lambda f x. x) f x))}_{\text{apply } \beta} \rightarrow_{\beta} \lambda f x. (f x)$$

So $\langle 1 \rangle g y \rightarrow_{\beta} (\lambda x. (g x)) y \rightarrow_{\beta} g y$

Church numerals . . .

$$\langle 2 \rangle = \lambda fx.f(\langle 1 \rangle fx) = \lambda fx.\underbrace{(f(\lambda fx.(fx)fx))}_{\text{apply } \beta} \rightarrow_{\beta} \lambda fx.(f(fx))$$

so,

$$\langle 2 \rangle gy \rightarrow_{\beta} \lambda x.(g(gx))y = g(gy)$$

Church numerals ...

$$\langle 2 \rangle = \lambda fx. f(\langle 1 \rangle fx) = \lambda fx. (f(\underbrace{\lambda fx. (fx) fx}_{\text{apply } \beta})) \rightarrow_{\beta} \lambda fx. (f(fx))$$

so,

$$\langle 2 \rangle gy \rightarrow_{\beta} \lambda x. (g(gx))y = g(gy)$$

- ▶ Let $g^k y$ denote $g(g(\dots(gy)))$ with k applications of g to y
- ▶ Show by induction that

$$\langle n \rangle = \lambda fx. f(\langle n-1 \rangle fx) \rightarrow_{\beta} \dots \rightarrow_{\beta} \lambda fx. (f^n x)$$

Encoding arithmetic functions ...

Successor

- ▶ $\text{succ}(n) = n + 1$
- ▶ Define as $\lambda pfx.f(pfx)$

Encoding arithmetic functions ...

Successor

- ▶ $\text{succ}(n) = n + 1$
- ▶ Define as $\lambda pfx.f(pfx)$

$$(\lambda pfx.f(pfx))\langle n \rangle$$

Encoding arithmetic functions ...

Successor

- ▶ $\text{succ}(n) = n + 1$
- ▶ Define as $\lambda pfx.f(pfx)$

$$(\lambda pfx.f(pfx))\langle n \rangle \rightarrow_{\beta} \lambda fx.f(\langle n \rangle fx)$$

Encoding arithmetic functions ...

Successor

- ▶ $\text{succ}(n) = n + 1$
- ▶ Define as $\lambda pfx.f(pfx)$

$$(\lambda pfx.f(pfx))\langle n \rangle \rightarrow_{\beta} \lambda fx.f(\langle n \rangle fx) \rightarrow_{\beta} \lambda fx.f(f^n x)$$

Encoding arithmetic functions ...

Successor

- ▶ $\text{succ}(n) = n + 1$
- ▶ Define as $\lambda pfx.f(pfx)$

$$(\lambda pfx.f(pfx))\langle n \rangle \rightarrow_{\beta} \lambda fx.f(\langle n \rangle fx) \rightarrow_{\beta} \lambda fx.f(f^n x) = \lambda fx.f^{n+1}x$$

Encoding arithmetic functions ...

Successor

- ▶ $\text{succ}(n) = n + 1$
- ▶ Define as $\lambda pfx.f(pfx)$

$$\begin{aligned} (\lambda pfx.f(pfx))\langle n \rangle &\rightarrow_{\beta} \lambda fx.f(\langle n \rangle fx) \rightarrow_{\beta} \lambda fx.f(f^n x) = \lambda fx.f^{n+1}x \\ &= \langle n+1 \rangle \end{aligned}$$

Encoding arithmetic functions ...

Successor

- ▶ $\text{succ}(n) = n + 1$
- ▶ Define as $\lambda pfx.f(pfx)$

$$\begin{aligned} (\lambda pfx.f(pfx))\langle n \rangle &\rightarrow_{\beta} \lambda fx.f(\langle n \rangle fx) \rightarrow_{\beta} \lambda fx.f(f^n x) = \lambda fx.f^{n+1}x \\ &= \langle n+1 \rangle \end{aligned}$$

plus: $\lambda pqfx.pf(qfx)$.

Encoding arithmetic functions ...

Successor

- ▶ $\text{succ}(n) = n + 1$
- ▶ Define as $\lambda pfx.f(pfx)$

$$\begin{aligned} (\lambda pfx.f(pfx))\langle n \rangle &\rightarrow_{\beta} \lambda fx.f(\langle n \rangle fx) \rightarrow_{\beta} \lambda fx.f(f^n x) = \lambda fx.f^{n+1}x \\ &= \langle n+1 \rangle \end{aligned}$$

plus: $\lambda pqfx.pf(qfx)$.