Programming Language Concepts: Lecture 13

S P Suresh

Chennai Mathematical Institute
spsuresh@cmi.ac.in
http://www.cmi.ac.in/~spsuresh/teaching/plc16

March 14, 2016

λ-calculus

- A notation for computable functions

λ-calculus

- A notation for computable functions
    - Alonzo Church

λ-calculus

- A notation for computable functions
    - Alonzo Church
- How do we describe a 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

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

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

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

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 countably infinite set *Var* of variables

λ-calculus: syntax

- Assume a countably infinite set Var of variables
- The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MN$$

where $x \in Var$ and $M, N \in \Lambda$.

λ-calculus: syntax

- Assume a countably infinite set Var of variables
- The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MN$$

  where $x \in Var$ and $M, N \in \Lambda$.
- $\lambda x.M$: Abstraction

λ-calculus: syntax

- Assume a countably infinite set *Var* of variables
- The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MN$$

  where $x \in Var$ and $M, N \in \Lambda$.
- $\lambda x.M$: Abstraction
  - A function of x with computation rule M.

λ-calculus: syntax

- Assume a countably infinite set Var of variables
- The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MN$$

  where $x \in Var$ and $M, N \in \Lambda$.
- $\lambda x.M$: Abstraction
  - A function of x with computation rule M.
  - "Abstracts" the computation rule M over arbitrary input values x

λ-calculus: syntax

- Assume a countably infinite set Var of variables
- The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MN$$

  where $x \in Var$ and $M, N \in \Lambda$.
- $\lambda x.M$: Abstraction
  - A function of x with computation rule M.
  - "Abstracts" the computation rule M over arbitrary input values x
  - Like writing $f(x) = e$, but not assigning a name f

λ-calculus: syntax

- Assume a countably infinite set Var of variables
- The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MN$$

  where $x \in Var$ and $M, N \in \Lambda$.
- $\lambda x.M$: Abstraction
  - A function of x with computation rule M.
  - "Abstracts" the computation rule M over arbitrary input values x
  - Like writing $f(x) = e$, but not assigning a name f
- MN: Application

λ-calculus: syntax

- Assume a countably infinite set Var of variables
- The set Λ of lambda expressions is given by

$$\Lambda = x \mid \lambda x.M \mid MN$$

  where $x \in \mathit{Var}$ and $M, N \in \Lambda$.
- $\lambda x.M$: Abstraction
  - A function of x with computation rule M.
  - "Abstracts" the computation rule M over arbitrary input values x
  - Like writing $f(x) = e$, but not assigning a name f
- MN: Application
  - Apply the function M to the argument N

- Can write expressions such as $x\,x$ — no types!

λ-calculus: syntax…

- Can write expressions such as $x\,x$ — no types!
- What can do without types?

λ-calculus: syntax…

- Can write expressions such as $x\,x$ — no types!
- What can do without types?
  - Set theory as a basis for mathematics

λ-calculus: syntax…

- Can write expressions such as $x\,x$ — no types!
- What can do without types?
  - Set theory as a basis for mathematics
  - Bit strings in memory

λ-calculus: syntax…

- Can write expressions such as $x\,x$ — no types!
- What can 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 $x x$ — no types!
- What can 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 yiled meaningful data

λ-calculus: syntax…

- Can write expressions such as $x\,x$ — no types!
- What can 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 yiled meaningful data
- Can also apply functions to non-meaningful data, but the result has no significance

λ-calculus: syntax…

- Application associates to the left

λ-calculus: syntax…

- Application associates to the left
    - $(MN)P$ is abbreviated MNP

λ-calculus: syntax…

- Application associates to the left
  - $(MN)P$ is abbreviated MNP
- Abstraction associates to the right

λ-calculus: syntax…

- Application associates to the left
  - $(MN)P$ is abbreviated MNP
- Abstraction associates to the right
  - $\lambda x.(\lambda y.M)$ is abbreviated $\lambda x.\lambda y.M$

λ-calculus: syntax…

- Application associates to the left
  - $(MN)P$ is abbreviated MNP
- Abstraction associates to the right
  - $\lambda x.(\lambda y.M)$ is abbreviated $\lambda x.\lambda y.M$
  - More drastically, $\lambda x_1.(\lambda x_2)\cdots(\lambda x_n.M)\cdots)$ is abbreviated $\lambda x_1 x_2 \cdots x_n.M$

λ-calculus: syntax…

- Application associates to the left
  - $(MN)P$ is abbreviated MNP
- Abstraction associates to the right
  - $\lambda x.(\lambda y.M)$ is abbreviated $\lambda x.\lambda y.M$
  - More drastically, $\lambda x_1.(\lambda x_2)\cdots(\lambda x_n.M)\cdots)$ is abbreviated $\lambda x_1 x_2 \cdots x_n.M$
  - $\lambda x.MN$ means $(\lambda x.(MN))$. Everything after the $\cdot$ is the body.

- Application associates to the left
  - $(MN)P$ is abbreviated MNP
- Abstraction associates to the right
  - $\lambda x.(\lambda y.M)$ is abbreviated $\lambda x.\lambda y.M$
  - More drastically, $\lambda x_1.(\lambda x_2)\cdots(\lambda x_n.M)\cdots)$ is abbreviated $\lambda x_1 x_2 \cdots x_n.M$
  - $\lambda x.MN$ means $(\lambda x.(MN))$. Everything after the $\cdot$ is the body.
  - Use $(\lambda x.M)N$ for applying $\lambda x.M$ to N

λ-calculus: syntax…

- Application associates to the left
  - $(MN)P$ is abbreviated MNP
- Abstraction associates to the right
  - $\lambda x.(\lambda y.M)$ is abbreviated $\lambda x.\lambda y.M$
  - More drastically, $\lambda x_1.(\lambda x_2)\cdots(\lambda x_n.M)\cdots)$ is abbreviated $\lambda x_1 x_2 \cdots x_n.M$
  - $\lambda x.MN$ means $(\lambda x.(MN))$. Everything after the $\cdot$ is the body.
  - Use $(\lambda x.M)N$ for applying $\lambda x.M$ to N
- Examples

- Application associates to the left
  - $(MN)P$ is abbreviated MNP
- Abstraction associates to the right
  - $\lambda x.(\lambda y.M)$ is abbreviated $\lambda x.\lambda y.M$
  - More drastically, $\lambda x_1.(\lambda x_2)\cdots(\lambda x_n.M)\cdots)$ is abbreviated $\lambda x_1 x_2 \cdots x_n.M$
  - $\lambda x.MN$ means $(\lambda x.(MN))$. Everything after the $\cdot$ is the body.
  - Use $(\lambda x.M)N$ for applying $\lambda x.M$ to N
- Examples
  - $(\lambda x.x)(\lambda y.y)(\lambda z.z)$ is short for $((\lambda x.x)(\lambda y.y))(\lambda z.z)$

λ-calculus: syntax…

- Application associates to the left
  - $(MN)P$ is abbreviated MNP
- Abstraction associates to the right
  - $\lambda x.(\lambda y.M)$ is abbreviated $\lambda x.\lambda y.M$
  - More drastically, $\lambda x_1.(\lambda x_2)\cdots(\lambda x_n.M)\cdots)$ is abbreviated $\lambda x_1 x_2 \cdots x_n.M$
  - $\lambda x.MN$ means $(\lambda x.(MN))$. Everything after the $\cdot$ is the body.
  - Use $(\lambda x.M)N$ for applying $\lambda x.M$ to N
- Examples
  - $(\lambda x.x)(\lambda y.y)(\lambda z.z)$ is short for $((\lambda x.x)(\lambda y.y))(\lambda z.z)$
  - $\lambda f.(\lambda u.f(uu))(\lambda u.f(uu))$ is short for $(\lambda f.((\lambda u.f(uu))(\lambda u.f(uu))))$

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_{\beta} M[x := N]$

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex
  - $M[x := N]$ is the contractum

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex
  - $M[x := N]$ is the contractum
- $M[x := N]$: substitute free occurrences of x in M by N

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex
  - $M[x := N]$ is the contractum
- $M[x := N]$: substitute free occurrences of x in M by N
- This is the normal rule we use for functions:

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex
  - $M[x := N]$ is the contractum
- $M[x := N]$: substitute free occurrences of x in M by N
- This is the normal rule we use for functions:
  - $f(x) = 2x^3 + 5x + 3$

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_{\beta} M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex
  - $M[x := N]$ is the contractum
- $M[x := N]$: substitute free occurrences of x in M by N
- This is the normal rule we use for functions:
  - $f(x) = 2x^3 + 5x + 3$
  - $f(7) = (2x^3 + 5x + 3)[x := 7] = 2 \cdot 7^3 + 5 \cdot 7 + 3 = 724$

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex
  - $M[x := N]$ is the contractum
- $M[x := N]$: substitute free occurrences of x in M by N
- This is the normal rule we use for functions:
  - $f(x) = 2x^3 + 5x + 3$
  - $f(7) = (2x^3 + 5x + 3)[x := 7] = 2 \cdot 7^3 + 5 \cdot 7 + 3 = 724$
- β is the only rule we need

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex
  - $M[x := N]$ is the contractum
- $M[x := N]$: substitute free occurrences of x in M by N
- This is the normal rule we use for functions:
  - $f(x) = 2x^3 + 5x + 3$
  - $f(7) = (2x^3 + 5x + 3)[x := 7] = 2 \cdot 7^3 + 5 \cdot 7 + 3 = 724$
- β is the only rule we need
- MN is meaningful only if M is of the form $\lambda x.P$

The computation rule β

- Basic rule for computation (rewriting) is called β-reduction (or contraction)
  - $(\lambda x.M)N \longrightarrow_\beta M[x := N]$
  - A term of the form $(\lambda x.M)N$ is a redex
  - $M[x := N]$ is the contractum
- $M[x := N]$: substitute free occurrences of x in M by N
- This is the normal rule we use for functions:
  - $f(x) = 2x^3 + 5x + 3$
  - $f(7) = (2x^3 + 5x + 3)[x := 7] = 2 \cdot 7^3 + 5 \cdot 7 + 3 = 724$
- β is the only rule we need
- MN is meaningful only if M is of the form $\lambda x.P$
  - Cannot do anything with terms like xx or $(y(\lambda x.x))(\lambda y.y)$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $BV(M)$: set of all variables occurring bound in M

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $BV(M)$: set of all variables occurring bound in M
  - $BV(x) = \emptyset$, for any $x \in Var$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $BV(M)$: set of all variables occurring bound in M
  - $BV(x) = \emptyset$, for any $x \in Var$
  - $BV(MN) = BV(M) \cup BV(N)$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $BV(M)$: set of all variables occurring bound in M
  - $BV(x) = \emptyset$, for any $x \in Var$
  - $BV(MN) = BV(M) \cup BV(N)$
  - $BV(\lambda x.M) = BV(M) \cup \{x\}$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $BV(M)$: set of all variables occurring bound in M
  - $BV(x) = \emptyset$, for any $x \in Var$
  - $BV(MN) = BV(M) \cup BV(N)$
  - $BV(\lambda x.M) = BV(M) \cup \{x\}$
- Example: $M = xy(\lambda x.z)(\lambda y.y)$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $BV(M)$: set of all variables occurring bound in M
  - $BV(x) = \emptyset$, for any $x \in Var$
  - $BV(MN) = BV(M) \cup BV(N)$
  - $BV(\lambda x.M) = BV(M) \cup \{x\}$
- Example: $M = xy(\lambda x.z)(\lambda y.y)$
  - $FV(M) = \{x, y, z\}$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $BV(M)$: set of all variables occurring bound in M
  - $BV(x) = \emptyset$, for any $x \in Var$
  - $BV(MN) = BV(M) \cup BV(N)$
  - $BV(\lambda x.M) = BV(M) \cup \{x\}$
- Example: $M = xy(\lambda x.z)(\lambda y.y)$
  - $FV(M) = \{x, y, z\}$
  - $BV(M) = \{y\}$

Free and bound variables

- An occurrence of a variable x in M is free if it does not occur in the scope of a λx inside M
- $FV(M)$: set of all variables occurring free in M
  - $FV(x) = \{x\}$, for any $x \in Var$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) \setminus \{x\}$
- $BV(M)$: set of all variables occurring bound in M
  - $BV(x) = \emptyset$, for any $x \in Var$
  - $BV(MN) = BV(M) \cup BV(N)$
  - $BV(\lambda x.M) = BV(M) \cup \{x\}$
- Example: $M = xy(\lambda x.z)(\lambda y.y)$
  - $FV(M) = \{x, y, z\}$
  - $BV(M) = \{y\}$
  - Warning: Possible for a variable to be both in $FV(M)$ and $BV(M)$

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N
  - Meaning of M: Take an argument and apply y to it!

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N
  - Meaning of M: Take an argument and apply y to it!
- β-reduction on M yields $\lambda y.yy$

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N
  - Meaning of M: Take an argument and apply y to it!
- β-reduction on M yields $\lambda y.yy$
  - Meaning: Take an argument and apply it to itself!

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N
  - Meaning of M: Take an argument and apply y to it!
- β-reduction on M yields $\lambda y.yy$
  - Meaning: Take an argument and apply it to itself!
- The y substituted for inner x has been "confused" with the y bound by λy

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N
  - Meaning of M: Take an argument and apply y to it!
- β-reduction on M yields $\lambda y.yy$
  - Meaning: Take an argument and apply it to itself!
- The y substituted for inner x has been "confused" with the y bound by λy
- Rename bound variables to avoid capture

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N
  - Meaning of M: Take an argument and apply y to it!
- β-reduction on M yields $\lambda y.yy$
  - Meaning: Take an argument and apply it to itself!
- The y substituted for inner x has been "confused" with the y bound by λy
- Rename bound variables to avoid capture
  - $(\lambda x.(\lambda y.xy))y = (\lambda x.(\lambda z.xz))y \longrightarrow_\beta \lambda z.yz$

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N
  - Meaning of M: Take an argument and apply y to it!
- β-reduction on M yields $\lambda y.yy$
  - Meaning: Take an argument and apply it to itself!
- The y substituted for inner x has been "confused" with the y bound by λy
- Rename bound variables to avoid capture
  - $(\lambda x.(\lambda y.xy))y = (\lambda x.(\lambda z.xz))y \longrightarrow_{\beta} \lambda z.yz$
- Renaming bound variables does not change the funciton

Variable capture

- Consider $N = \lambda x.(\lambda y.xy)$ and $M = Ny$
  - N takes two arguments and applies the first argument to the second
  - M fixes the first argument of N
  - Meaning of M: Take an argument and apply y to it!
- β-reduction on M yields $\lambda y.yy$
  - Meaning: Take an argument and apply it to itself!
- The y substituted for inner x has been "confused" with the y bound by λy
- Rename bound variables to avoid capture
  - $(\lambda x.(\lambda y.xy))y = (\lambda x.(\lambda z.xz))y \longrightarrow_{\beta} \lambda z.yz$
- Renaming bound variables does not change the funciton
  - $f(x) = 2x + 7$ vs $f(z) = 2z + 7$

$$M[x := N]$$

- $x[x := N] = N$

$M[x := N]$

- $x[x := N] = N$
- $y[x := N] = y$, where $y \in Var$ and $y \neq x$

$M[x := N]$

- $x[x := N] = N$
- $y[x := N] = y$, where $y \in Var$ and $y \neq x$
- $(PQ)[x := N] = (P[x := N])(Q[x := N])$

$M[x := N]$

- $x[x := N] = N$
- $y[x := N] = y$, where $y \in Var$ and $y \neq x$
- $(PQ)[x := N] = (P[x := N])(Q[x := N])$
- $(\lambda x.P)[x := N] = \lambda x.P$

$M[x := N]$

- $x[x := N] = N$
- $y[x := N] = y$, where $y \in Var$ and $y \neq x$
- $(PQ)[x := N] = (P[x := N])(Q[x := N])$
- $(\lambda x.P)[x := N] = \lambda x.P$
- $(\lambda y.P)[x := N] = \lambda y.(P[x := N])$, where $y \neq x$ and $y \notin FV(N)$

$M[x := N]$

- $x[x := N] = N$
- $y[x := N] = y$, where $y \in Var$ and $y \neq x$
- $(PQ)[x := N] = (P[x := N])(Q[x := N])$
- $(\lambda x.P)[x := N] = \lambda x.P$
- $(\lambda y.P)[x := N] = \lambda y.(P[x := N])$, where $y \neq x$ and $y \notin FV(N)$
- $(\lambda y.P)[x := N] = \lambda z.((P[y := z])[x := N])$, where $y \neq x$, $y \in FV(N)$, and z does not occur in P or N

$M[x := N]$

- $x[x := N] = N$
- $y[x := N] = y$, where $y \in \mathit{Var}$ and $y \neq x$
- $(PQ)[x := N] = (P[x := N])(Q[x := N])$
- $(\lambda x.P)[x := N] = \lambda x.P$
- $(\lambda y.P)[x := N] = \lambda y.(P[x := N])$, where $y \neq x$ and $y \notin FV(N)$
- $(\lambda y.P)[x := N] = \lambda z.((P[y := z])[x := N])$, where $y \neq x$, $y \in FV(N)$, and z does not occur in P or N
    - We fix a global ordering on Var and choose z to be the first variable not occurring in either P or N

$M[x := N]$

- $x[x := N] = N$
- $y[x := N] = y$, where $y \in Var$ and $y \neq x$
- $(PQ)[x := N] = (P[x := N])(Q[x := N])$
- $(\lambda x.P)[x := N] = \lambda x.P$
- $(\lambda y.P)[x := N] = \lambda y.(P[x := N])$, where $y \neq x$ and $y \notin FV(N)$
- $(\lambda y.P)[x := N] = \lambda z.((P[y := z])[x := N])$, where $y \neq x$, $y \in FV(N)$, and z does not occur in P or N
  - We fix a global ordering on Var and choose z to be the first variable not occurring in either P or N
  - Makes the definition deterministic

Applying β in context

- We can contract a redex appearing anywhere inside an expression

Applying β in context

- We can contract a redex appearing anywhere inside an expression
- Captured by the following rules

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

$$\frac{M \longrightarrow_\beta M'}{MN \longrightarrow_\beta M'N} \qquad \frac{N \longrightarrow_\beta N'}{MN \longrightarrow_\beta MN'} \qquad \frac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'}$$

Applying β in context

- We can contract a redex appearing anywhere inside an expression
- Captured by the following rules

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

$$\frac{M \longrightarrow_\beta M'}{MN \longrightarrow_\beta M'N} \qquad \frac{N \longrightarrow_\beta N'}{MN \longrightarrow_\beta MN'} \qquad \frac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'}$$

- $M \xrightarrow{*}_\beta N$: repeatedly apply β-reduction to get N

Applying β in context

- We can contract a redex appearing anywhere inside an expression
- Captured by the following rules

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

$$\frac{M \longrightarrow_\beta M'}{MN \longrightarrow_\beta M'N} \qquad \frac{N \longrightarrow_\beta N'}{MN \longrightarrow_\beta MN'} \qquad \frac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'}$$

- $M \stackrel{*}{\longrightarrow}_\beta N$: repeatedly apply β-reduction to get N
  - There is a sequence $M_0, M_1, \ldots, M_k$ such that

$$M = M_0 \longrightarrow_\beta M_1 \longrightarrow_\beta \cdots \longrightarrow_\beta M_k = N$$

Encoding arithmetic

- In set theory, use nesting to encode numbers

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0], [1], \ldots, [n-1]\}$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0], [1], \ldots, [n-1]\}$
  - Thus

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0],[1],\ldots,[n-1]\}$
  - Thus
    - $[0] = \emptyset$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0],[1],\ldots,[n-1]\}$
  - Thus
    - $[0] = \varnothing$
    - $[1] = \{\varnothing\}$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0], [1], \ldots, [n-1]\}$
  - Thus
    - $[0] = \varnothing$
    - $[1] = \{\varnothing\}$
    - $[2] = \{\varnothing, \{\varnothing\}\}$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0], [1], \ldots, [n-1]\}$
  - Thus
    - $[0] = \varnothing$
    - $[1] = \{\varnothing\}$
    - $[2] = \{\varnothing, \{\varnothing\}\}$
    - $[3] = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$

Encoding arithmetic

- In set theory, use nesting to encode numbers
  - Encoding of n: $[n]$
  - $[n] = \{[0], [1], \ldots, [n-1]\}$
  - Thus
    - $[0] = \emptyset$
    - $[1] = \{\emptyset\}$
    - $[2] = \{\emptyset, \{\emptyset\}\}$
    - $[3] = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$
- In λ-calculus, we encode n by the number of times we apply a function (successor) to an element (zero)

Church numerals

- $[n] = \lambda f x . f^n x$

Church numerals

- $[n] = \lambda f x. f^n x$
  - $f^0 x = x$

Church numerals

- $[n] = \lambda f x . f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$

Church numerals

- $[n] = \lambda f\, x. f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(f\, x)\cdots))$, where f is applied repeatedly n times

Church numerals

- $[n] = \lambda f x . f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- For instance

Church numerals

- $[n] = \lambda f x. f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda f x. x$

Church numerals

- $[n] = \lambda f x . f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda f x . x$
  - $[1] = \lambda f x . f x$

Church numerals

- $[n] = \lambda f x. f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda f x. x$
  - $[1] = \lambda f x. f x$
  - $[2] = \lambda f x. f(f x)$

Church numerals

- $[n] = \lambda f x. f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(f x)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda f x. x$
  - $[1] = \lambda f x. f x$
  - $[2] = \lambda f x. f(f x)$
  - $[3] = \lambda f x. f(f(f x))$

Church numerals

- $[n] = \lambda f\,x.f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(f\,x)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda f\,x.x$
  - $[1] = \lambda f\,x.f\,x$
  - $[2] = \lambda f\,x.f(f\,x)$
  - $[3] = \lambda f\,x.f(f(f\,x))$
  - ...

Church numerals

- $[n] = \lambda f x. f^n x$
  - $f^0 x = x$
  - $f^{n+1} x = f(f^n x)$
  - Thus $f^n x = f(f(\cdots(f\,x)\cdots))$, where f is applied repeatedly n times
- For instance
  - $[0] = \lambda f x. x$
  - $[1] = \lambda f x. f\,x$
  - $[2] = \lambda f x. f(f\,x)$
  - $[3] = \lambda f x. f(f(f\,x))$
  - ...
- $[n]\,g y = (\lambda f x. f(\cdots(f\,x)\cdots))g y \xrightarrow{*}_{\beta} g(\cdots(g y)\cdots) = g^n y$