Lambda Calculus

What is λ-calculus

- Programming language
  - Invented in 1930s, by Alonzo Church and Stephen Cole Kleene
- Model for computation
  - Alan Turing, 1937: Turing machines equal λ-calculus in expressiveness

Why learn λ-calculus

- Foundations of functional programming
  - Lisp, ML, Haskell, …
- Often used as a core language to study language theories
  - Type system
  - Scope and binding
  - Higher-order functions
  - Denotational semantics
  - Program equivalence
  - …

```
int x = 0;
for (int i = 0; i < 10; i++)  {  x++;  }
x = "abcd"; // bug (mistype)
i++;  // bug (out of scope)
```

How to formally define and rule out these bugs?

Overview: λ-calculus as a language

- Syntax
  - How to write a program?
  - Keyword "λ" for defining functions
- Semantics
  - How to describe the executions of a program?
  - Calculation rules called reduction
- Others
  - Type system (next class)
  - Model theory (not covered)
  - …

Syntax

- λ terms or λ expressions:

  (Terms) M, N ::= x | λx. M | M N

  - Lambda abstraction (λx.M): "anonymous" functions

    int f (int x) { return x; } ➔ λx. x

  - Lambda application (M N):

    int f (int x) { return x; }
    f(3); ➔ (λx. x) 3 = 3

Syntax

- λ terms or λ expressions:

  (Terms) $M, N ::= x \mid \lambda x.\ M \mid M\ N$

  - pure λ-calculus

- Add extra operations and data types
  - λx. (x+1)
  - λz. (x+2*y+z)
  - (λx. (x+1)) 3 = 3+1
  - (λz. (x+2*y+z)) 5 = x+2*y+5

Conventions

- Body of λ extends as far to the right as possible

  λx. M N means λx. (M N), **not** (λx. M) N

  - λx. f x = λx. (f x)
  - λx. λf. f x = λx. (λf. f x)

- Function applications are left-associative

  M N P means (M N) P, **not** M (N P)

  - (λx. λy. x - y) 5 3 = ((λx. λy. x - y) 5) 3
  - (λf. λx. f x) (λx. x + 1) 2 = ((λf. λx. f x) (λx. x + 1)) 2

Higher-order functions

- Functions can be returned as return values

$$\lambda x.\ \lambda y.\ x - y$$

- Functions can be passed as arguments

$$(\lambda f.\ \lambda x.\ f\ x)\ (\lambda x.\ x + 1)\ 2$$

Think about function pointers in C/C++.

Higher-order functions

- Given function f, return function $f \circ f$

$$\lambda f.\ \lambda x.\ f\ (f\ x)$$

- How does this work?

$$(\lambda f.\ \ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1)\ 5$$
$$=\ (\lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ x))\ 5$$
$$=\ (\lambda x.\ (\lambda y.\ y+1)\ (x+1))\ 5$$
$$=\ (\lambda x.\ (x+1)+1)\ 5$$
$$=\ 5+1+1 = 7$$

Curried functions

- Note difference between

$\lambda x.\ \lambda y.\ x - y$

and int f (int x, int y) { return x - y;}

- λ abstraction is a function of 1 parameter
- But computationally they are the same (can be transformed into each other)
  - Curry: transform $\lambda(x, y).\ x\text{-}y$ to $\lambda x.\ \lambda y.\ x - y$
  - Uncurry: the reverse of Curry

Free and bound variables

- $\lambda x.\ x + y$
  - x: bound variable
  - y: free variable

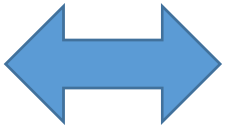
```
int y;
…
…
int add(int x) {
   return x + y;
}
```

Could be a global variable

Free and bound variables

- $\lambda x.\ x + y$
- Bound variable can be renamed ("placeholder")
  - $\lambda x.\ (x+y)$ is same function as $\lambda z.\ (z+y)$ α-equivalence
  - (x+y) is the ***scope*** of the binding λx

```
int add(int x) {
    return x + y;
}
x = 0; // out of scope!
```

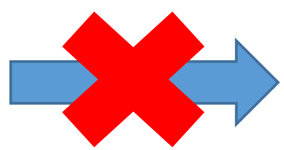


```
int add(int z) {
    return z + y;
}
```

Free and bound variables

- $\lambda x.\ x + y$
- Bound variable can be renamed ("placeholder")
  - $\lambda x.\ (x+y)$ is same function as $\lambda z.\ (z+y)$ α-equivalence
  - $(x+y)$ is the ***scope*** of the binding λx
- Name of free variable does matter
  - $\lambda x.\ (x+y)$ is *not* the same as $\lambda x.\ (x+z)$

```
int y = 10;
int z = 20;
int add(int x) {  return x + y;  }
```



```
int y = 10;
int z = 20;
int add(int x) {  return x + z;  }
```

Free and bound variables

- λx. x + y
- Bound variable can be renamed ("placeholder")
  - λx. (x+y) is same function as λz. (z+y) α-equivalence
  - (x+y) is the ***scope*** of the binding λx
- Name of free variable does matter
  - λx. (x+y) is *not* the same as λx. (x+z)
- Occurrences
  - (λx. x+y) (x+1) : x has both a free and a bound occurrence

```
int x = 10;
int add(int x) {   return x+y;}
add(x+1);
```

Formal definitions about free and bound variables

- Recall $M, N ::= x \mid \lambda x.\, M \mid M\ N$
- fv(M): the set of free variables in M

$$
\left.\begin{aligned}
fv(x) &\equiv \{x\} \\
fv(\lambda x.M) &\equiv fv(M) \setminus \{x\} \\
fv(M\ N) &\equiv fv(M) \cup fv(N)
\end{aligned}\right\} \text{Defined by induction on terms}
$$

- Example

$$fv((\lambda x.\ x)\ x) = \{x\}$$

$$fv((\lambda x.\ x + y)\ x) = \{x, y\}$$

Formal definitions about free and bound variables

- Recall $M, N ::= x \mid \lambda x.\ M \mid M\ N$
- fv(M): the set of free variables in M
- "x is a free variable in M": $x \in fv(M)$
- "x is a bound variable in M": ?
- α-equivalence: $\lambda x.\ M = \lambda y.\ M[y/x]$, where y fresh

Substitution (defined later)

Main points till now

- Syntax: notation for defining functions

  (Terms) $M, N ::= x \mid \lambda x.\, M \mid M\ N$

- Next: semantics (reduction rules)

Overview of reduction

- Basic rule is β-reduction

$$(\lambda x.\ M)\ N \quad \to \quad M[N/x]$$ ***(Substitution)***

- Repeatedly apply reduction rule to any sub-term

Example

$(\lambda f.\ \ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1)\ 5$

$\to\ (\lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ x))\ 5$

$\to\ (\lambda x.\ (\lambda y.\ y+1)\ (x+1))\ 5$

$\to\ (\lambda x.\ (x+1)+1)\ 5$

$\to\ 5+1+1 \to 7$

Substitution

- $M[N/x]$: replace x by N in M
  - Defined by induction on terms

$x[N/x] \equiv N$

$y[N/x] \equiv y$

$(M\ P)[N/x] \equiv (M[N/x])\ (P[N/x])$

$(\lambda x.M)[N/x] \equiv \lambda x.M$ ***(Only replace free variables!)***

$(\lambda y.M)[N/x] \equiv\ ?$

Because names of bound variables do ***not*** matter

Substitution – avoid name capture

- Example : $(\lambda x.\ x - y)[x/y]$

Substitute "blindly": $\lambda x.\ x - x$

Problem: unintended name capture!!

Solution: rename bound variables before substitution

$(\lambda x.\ x - y)[x/y]$

$= (\lambda z.\ z - y)[x/y]$

$= \lambda z.\ z - x$

Substitution – avoid name capture

- Example : $(\lambda x.\ f\ (f\ x))[(\lambda y.\ y+x)/f]$

Substitute "blindly": $\lambda x.\ (\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ x)$

Problem: x in $(\lambda y.\ y+x)$ got bound – unintended name capture!!

Solution: rename bound variables before substitution

$(\lambda x.\ f\ (f\ x))[(\lambda y.\ y+x)/f]$

$= (\lambda z.\ f\ (f\ z))[(\lambda y.\ y+x)/f]$

$= \lambda z.\ (\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ z)$

Substitution

- $M[N/x]$: replace x by N in M

$x[N/x] \equiv N$

$y[N/x] \equiv y$

$(M\ P)[N/x] \equiv (M[N/x])\ (P[N/x])$

$(\lambda x.M)[N/x] \equiv \lambda x.M$

$(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x])$, if $y \notin fv(N)$

$(\lambda y.M)[N/x] \equiv \lambda z.(M[z/y][N/x])$, if $y \in fv(N)$ and z fresh

z is unused

Easy rule: always rename variables to be distinct

Examples of substitution

$(\lambda x.\ (\lambda y.\ y\ z)\ (\lambda w.\ w)\ z\ x)[y/z]$

$(\lambda x.\ (\lambda y.\ y\ y)\ z\ x)[(f\ x)/z]$

Reduction rules

$$\frac{}{(\lambda x.M)N \;\rightarrow\; M[N/x]}\ (\beta)$$

$$\frac{M \rightarrow M'}{M\ N \rightarrow M'N}$$

$$\frac{N \rightarrow N'}{M\ N \rightarrow M\ N'}$$

$$\frac{M \rightarrow M'}{\lambda x.M \rightarrow \lambda x.M'}$$

Repeatedly apply (β) to any sub-term

Examples

$(\lambda f.\ f\ x)\ (\lambda y.\ y)$ // apply (β)

$\rightarrow (f\ x)[(\lambda y.\ y)/f]$

$= (\lambda y.\ y)\ x$ // apply (β)

$\rightarrow y[x/y]$

$= x$

$$\frac{}{(\lambda x.\, M)\, N \rightarrow M[N/x]}\ (\beta)$$

$$\frac{M \rightarrow M'}{M\, N \rightarrow M'\, N}$$

$$\frac{N \rightarrow N'}{M\, N' \rightarrow M\, N'}$$

$$\frac{M \rightarrow M'}{\lambda x.\, M \rightarrow \lambda x.\, M'}$$

Examples

$$\frac{}{(\lambda x.\, M)\, N \to M[N/x]}\ (\beta)$$

$$\frac{M \to M'}{M\, N \to M'\, N}$$

$$\frac{N \to N'}{M\, N' \to M\, N'}$$

$$\frac{M \to M'}{\lambda x.\, M \to \lambda x.\, M'}$$

(λy. λx. x - y) x // apply (β)

→ (λx. x - y)[x/y]

= λz. ((x - y)[z/x][x/y])

= λz. ((z - y)[x/y])

= λz. z - x

Examples

$$\frac{}{(\lambda x.\, M)\, N \to M[N/x]}\ (\beta)$$

$$\frac{M \to M'}{M\, N \to M'\, N}$$

$$\frac{N \to N'}{M\, N' \to M\, N'}$$

$$\frac{M \to M'}{\lambda x.\, M \to \lambda x.\, M'}$$

λx. (λy. y+1) x // 4th rule

→ λx. x+1

(λy. y+1) x // (β) rule

→ (y+1)[x/y]

= x+1

Examples

$$\frac{}{(\lambda x.\, M)\, N \to M[N/x]}\ (\beta)$$

$$\frac{M \to M'}{M\, N \to M'\, N}$$

$$\frac{N \to N'}{M\, N' \to M\, N'}$$

$$\frac{M \to M'}{\lambda x.\, M \to \lambda x.\, M'}$$

(λf. λz. f (f z)) (λy. y+x) // apply (β)

→ λz. (λy. y+x) ((λy. y+x) z) // apply (β) and the 3rd &4th rules

→ λz. (λy. y+x) (z+x) // apply (β) and the 4th rule

→ λz. z+x+x

Normal form

reducible expression

- β-redex: a term of the form (λx.M) N
- β-normal form: a term containing no β-redex
  - Stopping point: cannot further apply β-reduction rules

(λf. λx. f (f x)) (λy. y+1) 2

→ (λx. (λy. y+1) ((λy. y+1) x)) 2

→ (λx. (λy. y+1) (x+1)) 2

→ (λx. x+1+1) 2

→ 2+1+1 **(β-normal form)**

Can further reduce to 4 if having reduction rules for +

Confluence (Church-Rosser Property)

Expressions can be evaluated in any order.

Final result (if there is one) is uniquely determined.

$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1)\ 2$

$\rightarrow (\ \lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ x)\ )\ 2$

$\rightarrow (\ \lambda x.\ (\lambda y.\ y+1)\ (x+1)\ )\ 2$

$\rightarrow (\ \lambda x.\ x+1+1\ )\ 2$

$\rightarrow 2+1+1$

$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1)\ 2$

$\rightarrow (\ \lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ x)\ )\ 2$

$\rightarrow (\ \lambda x.\ (\lambda y.\ y+1)\ (x+1)\ )\ 2$

$\rightarrow (\lambda y.\ y+1)\ (2+1)$

$\rightarrow 2+1+1$

Formalizing Confluence Theorem

- $M \rightarrow^* M'$: zero-or-more steps of $\rightarrow$

  $M \rightarrow^0 M'$ iff $M = M'$

  $M \rightarrow^{k+1} M'$ iff $\exists M''.\ M \rightarrow M'' \wedge M'' \rightarrow^k M'$

  $M \rightarrow^* M'$ iff $\exists k.\ M \rightarrow^k M'$

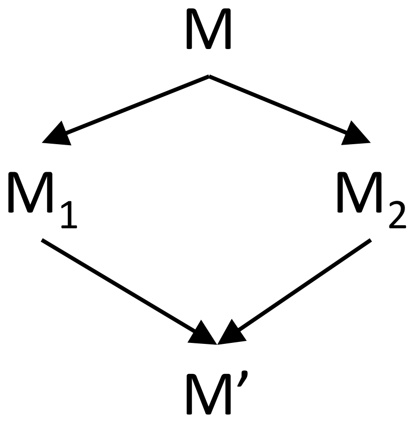
  inductive definition

- Confluence Theorem:

If $M \rightarrow^* M_1$ and $M \rightarrow^* M_2$,

then there exists M' such that

$M_1 \rightarrow^* M'$ and $M_2 \rightarrow^* M'$.



Non-terminating reduction

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\rightarrow \ldots$

$(\lambda x.\ x\ x\ y)\ (\lambda x.\ x\ x\ y)$

$\rightarrow (\lambda x.\ x\ x\ y)\ (\lambda x.\ x\ x\ y)\ y$

$\rightarrow \ldots$

$(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\rightarrow f\ \big((\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))\big)$

$\rightarrow \ldots$

Term may have both terminating and non-terminating reduction sequences

$(\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\rightarrow \lambda v.\ v$

$(\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\rightarrow (\lambda u.\ \lambda v.\ v)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

$\rightarrow \ldots$

Reduction strategies

- Normal-order reduction: choose the left-most, **outer-most** redex first

  (λu. λv. v) ((λx. x x)(λx. x x))

  → λv. v

  Normal-order reduction will find normal form if exists

- Applicative-order reduction: choose the left-most, **inner-most** redex first

  (λu. λv. v) ((λx. x x)(λx. x x))

  → (λu. λv. v) ((λx. x x)(λx. x x))

  → …

Reduction strategies – examples

Normal-order	***Applicative-order***
$(\lambda x.\ x\ x)\ ((\lambda y.\ y)\ (\lambda z.\ z))$	$(\lambda x.\ x\ x)\ ((\lambda y.\ y)\ (\lambda z.\ z))$
$\rightarrow ((\lambda y.\ y)\ (\lambda z.\ z))\ ((\lambda y.\ y)\ (\lambda z.\ z))$	$\rightarrow (\lambda x.\ x\ x)\ (\lambda z.\ z)$
$\rightarrow (\lambda z.\ z)\ ((\lambda y.\ y)\ (\lambda z.\ z))$	$\rightarrow (\lambda z.\ z)\ (\lambda z.\ z)$
$\rightarrow (\lambda y.\ y)\ (\lambda z.\ z)$	$\rightarrow \lambda z.\ z$
$\rightarrow \lambda z.\ z$	

Reduction strategies – examples

Applicative-order may ***not*** be as efficient as normal-order when the argument is not used.

Normal-order

$(\lambda x.\ p)\ ((\lambda y.\ y)\ (\lambda z.\ z))$

$\rightarrow p$

Applicative-order

$(\lambda x.\ p)\ ((\lambda y.\ y)\ (\lambda z.\ z))$

$\rightarrow (\lambda x.\ p)\ (\lambda z.\ z)$

$\rightarrow p$

Reduction strategies

- Similar to (but subtly different from) ***evaluation strategies*** in language theories
  - **Call-by-name** (like normal-order)
    - ALGOL 60

    arguments are not evaluated, but directly substituted into function body

  - **Call-by-need** (“memorized version” of call-by-name)
    - Haskell, R, …

    called “lazy evaluation”

  - **Call-by-value** (like applicative-order)
    - C, …

    called “eager evaluation”

  - …

Evaluation

- Only evaluate closed terms (i.e. no free variables)
- May not reduce all the way to a normal form
  - Terminate as soon as a *canonical form* (i.e. an abstraction) is obtained

$$
\begin{aligned}
(\lambda x.\, x(\lambda y.\, x\, y\, y)x)(\lambda z.\, \lambda w.\, z) &\to (\lambda z.\, \lambda w.\, z)(\lambda y.\, (\lambda z.\, \lambda w.\, z)y\, y)(\lambda z.\, \lambda w.\, z) \\
&\to (\lambda w.\, \lambda y.\, (\lambda z.\, \lambda w.\, z)y\, y)(\lambda z.\, \lambda w.\, z) \\
&\to \lambda y.\, (\lambda z.\, \lambda w.\, z)y\, y \\
&\to \lambda y.\, (\lambda w.\, y)y \\
&\to \lambda y.\, y.
\end{aligned}
$$

Evaluation terminates here

Evaluation

- A closed normal form must be a canonical form
- Not every closed canonical form is a normal form

- Recall that normal-order reduction will find the normal form if it exists
  - If normal-order reduction terminates, the reduction sequence must contain a first canonical form
  - Normal-order evaluation

Normal-order reduction & evaluation

- Normal-order reduction terminates

$$(\lambda x.\ \lambda y.\ x\,y)(\lambda x.\ x) \rightarrow \lambda y.\ (\lambda x.\ x)\,y \rightarrow \lambda y.\ y$$

Evaluation terminates here

- Normal-order reduction does not terminate

$$(\lambda x.\ \lambda y.\ x\,x)(\lambda x.\ x\,x) \rightarrow \lambda y.\ (\lambda x.\ x\,x)(\lambda x.\ x\,x) \rightarrow \lambda y.\ (\lambda x.\ x\,x)(\lambda x.\ x\,x) \rightarrow \cdots$$

Evaluation terminates here

$$(\lambda x.\ x\,x)(\lambda x.\ x\,x) \rightarrow (\lambda x.\ x\,x)(\lambda x.\ x\,x) \rightarrow \cdots$$

Evaluation diverges too

Normal-order evaluation rules

$$\frac{}{\lambda x. M \Rightarrow \lambda x. M} \text{ (Term)}$$

$$\frac{M \Rightarrow \lambda x. M' \qquad M'[N/x] \Rightarrow P}{M\ N \ \Rightarrow \ P} \ (\beta)$$

Normal-order evaluation – example

$(\lambda x.\ x(\lambda y.\ x\,y\,y)x)(\lambda z.\ \lambda w.\ z)$

$\quad\lambda x.\ x(\lambda y.\ x\,y\,y)x \Rightarrow \lambda x.\ x(\lambda y.\ x\,y\,y)x$

$\quad(\lambda z.\ \lambda w.\ z)(\lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y)(\lambda z.\ \lambda w.\ z)$

$\quad\quad(\lambda z.\ \lambda w.\ z)(\lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y)$

$\quad\quad\quad\lambda z.\ \lambda w.\ z \Rightarrow \lambda z.\ \lambda w.\ z$

$\quad\quad\quad\lambda w.\ \lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y \Rightarrow \lambda w.\ \lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y$

$\quad\quad\Rightarrow \lambda w.\ \lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y$

$\quad\quad\lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y \Rightarrow \lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y$

$\quad\Rightarrow \lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y$

$\Rightarrow \lambda y.\ (\lambda z.\ \lambda w.\ z)y\,y.$

Recall the reduction strategies

Normal-order

$(\lambda x.\ x\ x)\ ((\lambda y.\ y)\ (\lambda z.\ z))$

$\rightarrow ((\lambda y.\ y)\ (\lambda z.\ z))\ ((\lambda y.\ y)\ (\lambda z.\ z))$

$\rightarrow (\lambda z.\ z)\ ((\lambda y.\ y)\ (\lambda z.\ z))$

$\rightarrow (\lambda y.\ y)\ (\lambda z.\ z)$

$\rightarrow \lambda z.\ z$

Applicative-order

$(\lambda x.\ x\ x)\ ((\lambda y.\ y)\ (\lambda z.\ z))$

$\rightarrow (\lambda x.\ x\ x)\ (\lambda z.\ z)$

$\rightarrow (\lambda z.\ z)\ (\lambda z.\ z)$

$\rightarrow \lambda z.\ z$

Eager evaluation:
Postpone the substitution until the argument is a canonical form. No need to reduce many copies of the argument separately.

Eager evaluation rules

$$\frac{}{\lambda x. M \Rightarrow_E \lambda x. M} \text{ (Term)}$$

$$\frac{M \Rightarrow_E \lambda x. M' \qquad N \Rightarrow_E N' \qquad M'[N'/x] \Rightarrow_E P}{M\ N \Rightarrow_E P} \text{ (β)}$$

Eager evaluation – example

$$(\lambda x.\ x\,x)((\lambda y.\ y)(\lambda z.\ z))$$

$$\lambda x.\ x\,x \Rightarrow_E \lambda x.\ x\,x$$

$$(\lambda y.\ y)(\lambda z.\ z)$$

$$\lambda y.\ y \Rightarrow_E \lambda y.\ y$$

$$\lambda z.\ z \Rightarrow_E \lambda z.\ z$$

$$\lambda z.\ z \Rightarrow_E \lambda z.\ z$$

$$\Rightarrow_E \lambda z.\ z$$

$$(\lambda z.\ z)(\lambda z.\ z)$$

$$\lambda z.\ z \Rightarrow_E \lambda z.\ z$$

$$\lambda z.\ z \Rightarrow_E \lambda z.\ z$$

$$\lambda z.\ z \Rightarrow_E \lambda z.\ z$$

$$\Rightarrow_E \lambda z.\ z$$

$$\Rightarrow_E \lambda z.\ z.$$

Main points till now

- Syntax: notation for defining functions

$$(\text{Terms})\ \ M, N \ ::= \ x \ \mid \ \lambda x.\, M \ \mid \ M\, N$$

- Semantics (reduction rules)

$$(\lambda x.\, M)\, N \ \rightarrow \ M[N/x] \qquad (\beta)$$

- Next: programming in λ-calculus
  - Encoding data and operators in "pure" λ-calculus (without adding any additional syntax)

Programming in λ-calculus

- Encoding Boolean values and operators
  - True $\equiv$ λx. λy. x
  - False $\equiv$ λx. λy. y

Programming in λ-calculus

- Encoding Boolean values and operators
  - True $\equiv$ λx. λy. x
  - False $\equiv$ λx. λy. y
  - not $\equiv$ λb. b False True

not True
$\to$ True False True
$\to$ False

not False
$\to$ False False True
$\to$ True

Programming in λ-calculus

- Encoding Boolean values and operators
  - True $\equiv$ λx. λy. x
  - False $\equiv$ λx. λy. y
  - not $\equiv$ λb. b False True
  - and $\equiv$ λb. λb'. b b' False

and True b
$\to^*$ True b False
$\to$ b

and False b
$\to^*$ False b False
$\to$ False

Programming in λ-calculus

- Encoding Boolean values and operators
  - True $\equiv$ λx. λy. x
  - False $\equiv$ λx. λy. y
  - not $\equiv$ λb. b False True
  - and $\equiv$ λb. λb'. b b' False
  - or $\equiv$ λb. λb'. b True b'

or True b
$\to^*$ True True b
$\to$ True

or False b
$\to^*$ False True b
$\to$ b

Programming in λ-calculus

- Encoding Boolean values and operators
  - True $\equiv$ $\lambda x.\ \lambda y.\ x$
  - False $\equiv$ $\lambda x.\ \lambda y.\ y$
  - not $\equiv$ $\lambda b.$ b False True
  - and $\equiv$ $\lambda b.\ \lambda b'.$ b b' False
  - or $\equiv$ $\lambda b.\ \lambda b'.$ b True b'
  - if b then M else N $\equiv$ b M N

Not unique encoding

Programming in λ-calculus

- Encoding Boolean values and operators
  - True $\equiv$ $\lambda x.\ \lambda y.\ x$
  - False $\equiv$ $\lambda x.\ \lambda y.\ y$
  - not $\equiv$ $\lambda b.\ b$ False True
  - and $\equiv$ $\lambda b.\ \lambda b'.\ b\ b'$ False
  - or $\equiv$ $\lambda b.\ \lambda b'.\ b$ True b'
  - if b then M else N $\equiv$ b M N
  - not' $\equiv$ $\lambda b.\ \lambda x.\ \lambda y.\ b\ y\ x$

not' True
$\rightarrow \lambda x.\ \lambda y.$ True $y\ x$
$\rightarrow \lambda x.\ \lambda y.\ y$ = False

not' False
$\rightarrow \lambda x.\ \lambda y.$ False $y\ x$
$\rightarrow \lambda x.\ \lambda y.\ x$ = True

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\ \lambda x.\ x$ *(the same as False!)*
  - $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
  - $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\ \lambda x.\ x$
  - $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
  - $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$
  - $\text{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$

succ $\underline{n}$
$\to \lambda f.\ \lambda x.\ f\ (\underline{n}\ f\ x)$
$= \lambda f.\ \lambda x.\ f\ ((\lambda f.\ \lambda x.\ f^n\ x)\ f\ x)$
$\to \lambda f.\ \lambda x.\ f\ (f^n\ x)$
$= \lambda f.\ \lambda x.\ f^{n+1}\ x$
$= \underline{n+1}$

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\ \lambda x.\ x$
  - $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
  - $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$
  - $\text{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
  - $\text{succ'} \equiv \lambda n.\ \lambda f.\ \lambda x.\ n\ f\ (f\ x)$

$\text{succ'}\ \underline{n} \to^* \underline{n+1}$

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\ \lambda x.\ x$
  - $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
  - $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$
  - $\text{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
  - $\text{iszero} \equiv \lambda n.\ \lambda x.\ \lambda y.\ n\ (\lambda z.\ y)\ x$

iszero $\underline{0}$
$\to \lambda x.\ \lambda y.\ \underline{0}\ (\lambda z.\ y)\ x$
$= \lambda x.\ \lambda y.\ (\lambda f.\ \lambda x.\ x)\ (\lambda z.\ y)\ x$
$\to \lambda x.\ \lambda y.\ (\lambda x.\ x)\ x$
$\to \lambda x.\ \lambda y.\ x$ = True

iszero $\underline{1}$
$\to \lambda x.\ \lambda y.\ \underline{1}\ (\lambda z.\ y)\ x$
$= \lambda x.\ \lambda y.\ (\lambda f.\ \lambda x.\ f\ x)\ (\lambda z.\ y)\ x$
$\to \lambda x.\ \lambda y.\ (\lambda x.\ (\lambda z.\ y)\ x)\ x$
$\to \lambda x.\ \lambda y.\ ((\lambda z.\ y)\ x)$
$\to \lambda x.\ \lambda y.\ y$ = False

iszero (succ $\underline{n}$) $\to^*$ False

Programming in λ-calculus

- Church numerals
  - $\underline{0} \equiv \lambda f.\ \lambda x.\ x$
  - $\underline{1} \equiv \lambda f.\ \lambda x.\ f\ x$
  - $\underline{2} \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
  - $\underline{n} \equiv \lambda f.\ \lambda x.\ f^n\ x$
  - succ $\equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
  - iszero $\equiv \lambda n.\ \lambda x.\ \lambda y.\ n\ (\lambda z.\ y)\ x$
  - add $\equiv \lambda n.\ \lambda m.\ \lambda f.\ \lambda x.\ n\ f\ (m\ f\ x)$
  - mult $\equiv \lambda n.\ \lambda m.\ \lambda f.\ n\ m\ f$

Programming in λ-calculus

- Pairs
  - $(M, N) \equiv \lambda f.\ f\ M\ N$
  - $\pi_0 \equiv \lambda p.\ p\ (\lambda x.\ \lambda y.\ x)$
  - $\pi_1 \equiv \lambda p.\ p\ (\lambda x.\ \lambda y.\ y)$

$\pi_0(M, N) \rightarrow^* M$

$\pi_1(M, N) \rightarrow^* N$

Programming in λ-calculus

- Pairs
  - $(M, N) \equiv \lambda f.\ f\ M\ N$
  - $\pi_0 \equiv \lambda p.\ p\ (\lambda x.\ \lambda y.\ x)$
  - $\pi_1 \equiv \lambda p.\ p\ (\lambda x.\ \lambda y.\ y)$
- Tuples
  - $(M_1, \ldots, M_n) \equiv \lambda f.\ f\ M_1 \ldots M_n$
  - $\pi_i \equiv \lambda p.\ p\ (\lambda x_1. \ldots \lambda x_n.\ x_i)$

Programming in λ-calculus

- Recursive functions
  - fact(n) = if (n == 0) then 1 else n * fact(n-1)
  - To find fact, we need to solve an equation!

Fixpoint in arithmetic

- x is a fixpoint of f   if   f(x) = x

- Some functions has fixpoints, while others don’t
  - f(x) = x * x. Two fixpoints 0 and 1.
  - f(x) = x + 1. No fixpoint.
  - f(x) = x. Infinitely many fixpoints.

fact is a fixpoint of a function

- x is a fixpoint of f   if   f(x) = x

fact(n) =   if (n == 0) then 1 else n * fact(n-1)

fact = λn. if (n == 0) then 1 else n * fact(n-1)

fact = (λf. λn. if (n == 0) then 1 else n * f(n-1)) fact

Let F = λf. λn. if (n == 0) then 1 else n * f(n-1).
Then   fact = F fact.   So fact is a fixpoint of F.

In λ-calculus, every term has a fixpoint

- Fixpoint combinator is a higher-order function h satisfying

  for all f, (h f) gives a fixpoint of f

  i.e. h f = f (h f)

  - Turing's fixpoint combinator Θ

    Let A = λx. λy. y (x x y) and Θ = A A

  - Church's fixpoint combinator **Y**

    Let **Y** = λf. (λx. f (x x)) (λx. f (x x))

Turing's fixpoint combinator Θ

- Let $A = \lambda x. \lambda y. y\ (x\ x\ y)$ and $\Theta = A\ A$
- Let's prove: for all f, $\Theta\ f = f\ (\Theta\ f)$

Solving fact

Let F = λf. λn. if (n == 0) then 1 else n * f(n-1).

fact is a fixpoint of F.

fact = Θ F

The right-hand side is a closed lambda term that represents the factorial function.

Comments on computability

Turing's Turing machine, Church's λ-calculus and Gödel's general recursive functions are equivalent to each other in the sense that they define the same class of functions (a.k.a computable functions).

This is proved by Church, Kleene, Rosser, and Turing.

Programming in λ-calculus

- Booleans
- Natural numbers
- Pairs
- Lists
- Trees
- Recursive functions
- …

Read supplementary materials on course website

Main points about λ-calculus

- Succinct function expressions
  - λ
  - Bound variables can be renamed
- Reduction via substitution
- Can be extended with
  - Types (next class)
  - Side-effects (not covered)