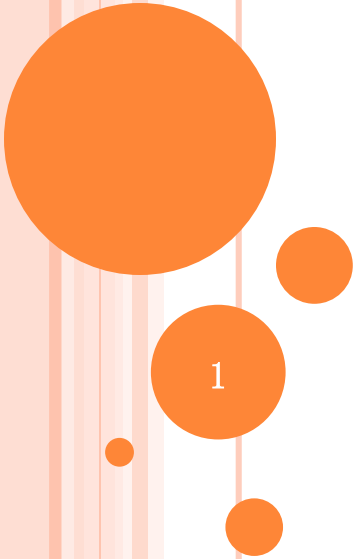


Untyped Lambda Calculus

Original λ-calculus Syntax

e is a *lambda expression*, or *lambda term*.

e ::=	x	(a variable)
	| \x.e	(a nameless function/lambda abstraction)
	| e_1 e_2	(function application)
v ::=	\x.e	(only functions can be values)

Above is a BNF (Backus Naur Form) that specifies the abstract syntax of the language

["\" will be written "λ" in a nice font]

Note the above is *inductive* definition: e, x are *meta-variables*

Functions

- Essentially every full-scale programming language has some notion of function
  - the (pure) lambda calculus is a language composed entirely of functions
  - we use the lambda calculus to study the essence of computation
  - it is just as fundamental as Turing Machines

More syntax

- the identity function:
  - \x.x
    - Mathematically equivalent to: $f(x) = x$.
- 2 notational conventions:
  - applications associate to the left (like in):
    - “y z x” is “(y z) x”
  - the body of a lambda abstraction extends as far as possible to the right:
    - “\x.x \z.x z x” is “\x.(x \z.(x z x))”

Names and Denotable Objects

- Name is a sequence of characters used to represent or *denote* an syntactic object.
- "Object" is used in the general sense. The most common object we see in this course is a variable.
- E.g.,

  \foo.foo \bar.foo bar foo

Names and Denotable Objects

- A name and the object it denotes are NOT the same thing!
- A name is merely a "*character string*".
- An object can have multiple names - "*aliasing*".
- A name can denote different objects at different times.
- "variable *bar*" means "the variable with the name *bar*".
- "function *foo*" means "the function with the name *foo*".

Binding

- *Binding* is an association between a name and the denotable object it represents
  - *Static binding*: during language design, compile time
  - *Dynamic binding*: during run time
- The *scope* of a name is the region of a program which can access the name binding.
- The *lifetime* of a name refers to the time interval (at runtime) during which the name remains *bound*.

Scopes in λ-calculus

- \x. e

x is the formal param of the function.
the scope of x is the term e (e is a meta-variable, meaning you can replace e with any valid lambda expression)

- \x. x y

y is *free* in the term \x. x y
i.e., y is not declared but used.

x is *bound*
in the term \x. x y

- λ-calculus uses static binding

Free Variables

- free (x) = x
- free(e1 e2) = free(e1) È free(e2)
- free (\x. e) = free(e) - {x}

Free Variables (Inductive Rules)

$$\frac{}{FV(x) = \{x\}}$$

$$\frac{FV(e1) = S1 \quad FV(e2) = S2}{FV(e1\ e2) = S1 \cup S2}$$

$$\frac{FV(e) = S}{FV(\backslash x.\, e) = S - \{x\}}$$

All Variables

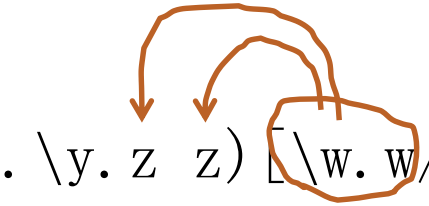
Vars(x) = {x}

Vars(e1 e2) = Vars(e1) ∪ Vars(e2)

Vars(\x.e) = Vars(e) ∪ {x}

Substitution

- e[v/x] is the term in which all *free* occurrences of *x* in *e* are replaced with *v*.
- this replacement operation is called *substitution*.



(\x.\y.z z)[\w.w/z] = \x.\y.(\w.w) (\w.w)

(\x.\z.z z)[\w.w/z] = \x.\z.z z

(\x.x z)[x/z] = ~~\x.x x~~

Capturing!

(\x.x z)[x/z] = (\y.y z)[x/z] = \y.y x

alpha-equivalent expressions = the same except for consistent renaming of bound variables

This process is also called alpha-renaming or alpha-reduction

"Special" substitution (ignoring capture issues)

Definition of e1 [[e/x]] assuming FV(e) ∩ Vars(e1) = ∅:

$$
\begin{array}{lll}
x\ [[e/x]] & = & e \\
y\ [[e/x]] & = & y \qquad\qquad (\text{if } y \neq x) \\
e1\ e2\ [[e/x]] & = & (e1\ [[e/x]])\ (e2\ [[e/x]]) \\
(\backslash x.e1)\ [[e/x]] & = & \backslash x.e1 \\
(\backslash y.e1)\ [[e/x]] & = & \backslash y.(e1\ [[e/x]]) \quad (\text{if } y \neq x)
\end{array}
$$

Alpha-Equivalence

In order to avoid variable clashes, it is very convenient to alpha-rename expressions so that bound variables don't get in the way.

eg: to alpha-rename \x.e we:

1. pick z such that z not in Vars(\x.e)
2. return \z.(e[[z/x]])

We previously defined e[[z/x]] in such a way that it is a total function when z is not in Vars(\x.e)

Terminology: Expressions e1 and e2 are called alpha-equivalent when they are the same after alpha-converting some of their bound variables

SUBSTITUTION (OFFICIAL)

$$
\begin{aligned}
x\ [e/x] &= e \\
y\ [e/x] &= y \qquad\qquad (\text{if } y \neq x) \\
e1\ e2\ [e/x] &= (e1\ [e/x])\ (e2\ [e/x])
\end{aligned}
$$

$$
(\backslash x.e1)[e/x] = \backslash x.e1
$$

$$
\begin{aligned}
(\backslash y.e1)[e/x] &= \backslash y.(e1[e/x]) \qquad (\text{if } y \neq x \ \&\ y \notin FV(e)) \\
&= \backslash z.(e1[[z/y]][e/x]) \\
&\qquad \text{pick } z \notin FV(e) (\text{if } y \neq x \ \&\ y \in FV(e))
\end{aligned}
$$

Operational Semantics

- single-step evaluation (judgment form): e → e'
- primary rule (beta reduction):

$$\frac{}{(\backslash x.e1)\ \ e2 \rightarrow e1\ [e2/x]}$$

- A term of the form (\x.e1) e2 is called redex (reducible expression).

Evaluation Strategies

- let id = \x. x, consider following exp with 3 redexes:

  id (id (\z. id z))

  id (id (\z. id z))

  id (id (\z. id z))

- Each strategy defines which redex in an expression gets reduced (fired) on the *next* step of evaluation
- *Full beta-reduction*: any redex

  id (id (\z. id z))

→ id (id (\z. z))

→ id (\z. z)

→ \z. z

Evaluation Strategies

- *Normal order*: leftmost, outermost redex first

  id (id (\z. id z))

→ id (\z. id z)

→ \z. id z

→ \z. z

- *Call-by-name*: similar to normal order except NO reduction inside lambda abstractions

  id (id (\z. id z))

→ id (\z. id z)

→ \z. id z

Evaluation Strategies

- *Call-by-value*: only outermost redex, whose RHS must be a value, no reduction inside abstraction
  - values are v ::= \x.e (lambda abstractions)

id (id (\z. id z))

→ id (\z. id z)

→ \z. id z

Another Example (diff between call by name and call by value)

- Call by name:

  (\x. y) ((\x. x x) (\x. x x))

→ y

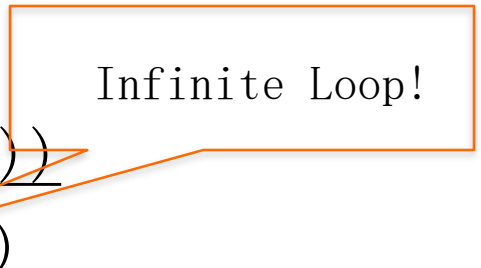
- Call by value:

  (\x. y) ((\x. x x) (\x. x x))

→ (\x. y) ((\x. x x) (\x. x x))

→ (\x. y) ((\x. x x) (\x. x x))

→ …



Call-by-value operational semantics

- Basic rule

$$\frac{}{(\backslash x.\, e)\quad v \rightarrow e\ [v/x]}$$

- Search rules:

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ \ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'}$$

- Notice, evaluation is left to right

ALTERNATIVES

$$\frac{}{(\backslash x.\, e)\ v \rightarrow e\ [v/x]}$$

$$\frac{}{(\backslash x.\, e1)\ e2 \rightarrow e1\ [e2/x]}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

$$\frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'}$$

call-by-value

call-by-name

ALTERNATIVES

$$\frac{}{(\backslash x.\, e)\ v \rightarrow e\ [v/x]}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

$$\frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'}$$

call-by-value

$$\frac{}{(\backslash x.\, e1)\ e2 \rightarrow e1\ [e2/x]}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

$$\frac{e \rightarrow e'}{\backslash x.\, e \rightarrow \backslash x.\, e'}$$

normal order

ALTERNATIVES

$$\frac{}{(\backslash x.\, e)\ v \rightarrow e\ [v/x]}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ \ e2}$$

$$\frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'}$$

call-by-value

$$\frac{}{(\backslash x.\, e1)\ e2 \rightarrow e1\ [e2/x]}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ \ e2}$$

$$\frac{e2 \rightarrow e2'}{e1\ e2 \rightarrow e1\ e2'}$$

$$\frac{e \rightarrow e'}{\backslash x.\, e \rightarrow \backslash x.\, e'}$$

full beta-reduction

ALTERNATIVES

$$\frac{}{(\backslash x.\, e)\ v \rightarrow e\ [v/x]}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

$$\frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'}$$

call-by-value

$$\frac{}{(\backslash x.\, e)\ v \rightarrow e\ [v/x]}$$

$$\frac{e1 \rightarrow e1'}{e1\ v \rightarrow e1'\ v}$$

$$\frac{e2 \rightarrow e2'}{e1\ e2 \rightarrow e1\ e2'}$$

right-to-left call-by-value

Proving Theorems About O.S.

Call-by-value o.s.:

$$\frac{}{(\backslash x.e)\ v \rightarrow e\ [v/x]} \qquad \frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'}$$

To prove property P of e1 → e2, there are 3 cases:

case:

$$\frac{}{(\backslash x.e)\ v \rightarrow e\ [v/x]}$$

Must prove: P((\x.e) v → e [v/x])
** Often requires a related property of substitution e[v/x]

case:

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

IH = P(e1 → e1')
Must prove: P(e1 e2 → e1' e2)

case:

$$\frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'}$$

IH = P(e2 → e2')
Must prove: P(v e2 → v e2')

Multi-step Op. Semantics

- Given a single step op sem. relation:

$$e1 \rightarrow e2$$

- We extend it to a multi-step relation by taking its "reflexive, transitive closure:"

$$\frac{}{e1 \rightarrow^* e1}\ \text{(reflexivity)} \qquad \frac{e1 \rightarrow e2 \quad e2 \rightarrow^* e3}{e1 \rightarrow^* e3}\ \text{(transitivity)}$$

Proving Theorems About O.S.

Call-by-value o.s.:

$$\frac{}{e1 \rightarrow^* e1} \quad \text{(reflexivity)} \qquad \frac{e1 \rightarrow e2 \quad e2 \rightarrow^* e3}{e1 \rightarrow^* e3} \quad \text{(transitivity)}$$

To prove property P of $e1 \rightarrow^* e2$, given you've already proven property P' of $e1 \rightarrow e2$, there are 2 cases:

case: $\frac{}{e1 \rightarrow^* e1}$

Must prove: $P(e1 \rightarrow^* e1)$ directly

case: $\frac{e1 \rightarrow e2 \quad e2 \rightarrow^* e3}{e1 \rightarrow^* e3}$

IH = $P(e2 \rightarrow^* e3)$

Also available: $P'(e1 \rightarrow e2)$

Must prove: $P(e1 \rightarrow^* e3)$

Example

Definition: An expression e is closed if FV(e) = { }.

Theorem:

If e1 is closed and e1 →* e2 then e2 is closed.

Proof: by induction on derivation of e1 →* e2.

(We need to prove lemma: if e1 is closed and e1 → e2, then e2 is closed.)