

UNTYPED LAMBDA CALCULUS

Original λ -CALCULUS SYNTAX

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

$e ::= x$	(a variable)
$\backslash x.e$	(a nameless function/ <i>lambda abstraction</i>)
$e_1 e_2$	(function application)

$v ::= \backslash x.e$ (only functions can be values)

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

[“ \backslash ” will be written “ λ ” in a nice font]

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

QUIZ

- In the following definition, list all the symbols that are meta variables

$e ::= x$	(a variable)
$\lambda x.e$	(a nameless function/ lambda abstraction)
$e_1 e_2$	(function application)

- Suppose we define a judgment form:

- e term

Can you re-define the lambda-term using the above judgment form and a few inference rules (using our good old axiom/proper rule format)?

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:
 - $\lambda x.x$
 - 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:
 - “ $\lambda x.x \ \lambda z.x \ z \ x$ ” is “ $\lambda x.(x \ \lambda z.(x \ z \ x))$ ”

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*”.


QUIZ

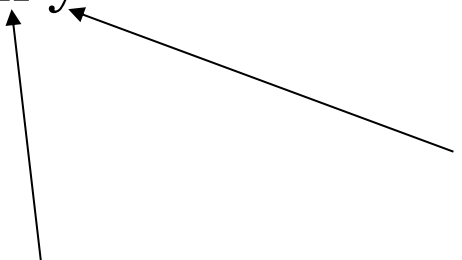
- Name one thing/object in computing, or in life that is NOT denotable?

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

- $\backslash x.e$ 

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)
- $\backslash x.x y$ 

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

y is *free* in the term $\backslash x.x y$
i.e., y is not declared but used.
- λ -calculus uses static binding

FREE VARIABLES

- $\text{free}(x) = x$
- $\text{free}(e1\ e2) = \text{free}(e1) \cup \text{free}(e2)$
- $\text{free}(\backslash x.e) = \text{free}(e) - \{x\}$

Judgement form?

$$\text{free}(e) = \{x\}$$

FREE VARIABLES (INFERENCE RULES)

$$\overline{\text{free}(x) = \{x\}}$$

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

$$\frac{\text{free}(e) = S}{\text{free}(\backslash x.e) = S - \{x\}}$$

ALL VARIABLES

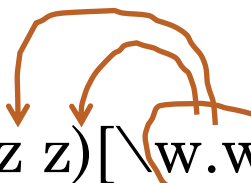
$$\text{Vars}(x) = \{x\}$$

$$\text{Vars}(e1 \ e2) = \text{Vars}(e1) \cup \text{Vars}(e2)$$

$$\text{Vars}(\backslash x.e) = \text{Vars}(e) \cup \{x\}$$

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*.


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

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

$$(\lambda x.x z)[x/z] = \underline{\lambda x.x x}$$

Capturing!

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


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 $e1 \llbracket e/x \rrbracket$ assuming $FV(e) \cap Vars(e1) = \emptyset$:

$$\begin{aligned}x \llbracket e/x \rrbracket &= e \\y \llbracket e/x \rrbracket &= y \quad (\text{if } y \neq x) \\e1 \ e2 \llbracket e/x \rrbracket &= (e1 \llbracket e/x \rrbracket) (e2 \llbracket e/x \rrbracket) \\(\lambda x.e1) \llbracket e/x \rrbracket &= \lambda x.e1 \\(\lambda y.e1) \llbracket e/x \rrbracket &= \lambda y.(e1 \llbracket e/x \rrbracket) \quad (\text{if } y \neq x)\end{aligned}$$

ALPHA-EQUIVALENCE

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

eg: to alpha-rewrite $\lambda x.e$ we:

1. pick z such that $z \text{ not in } \text{Vars}(\lambda x.e)$
2. return $\lambda z.(e[[z/x]])$

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

Terminology: Expressions e_1 and e_2 are called **alpha-equivalent** when they are the same after alpha-converting some of their bound variables

SUBSTITUTION (OFFICIAL)

$$x [e/x] = e$$

$$y [e/x] = y \quad (\text{if } y \neq x)$$

$$e1 \ e2 [e/x] = (e1 [e/x]) (e2 [e/x])$$

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

$$(\lambda y. e1)[e/x] = \lambda y. (e1 [e/x]) \quad (\text{if } y \neq x \ \& \ y \notin FV(e))$$

$$= \lambda z. (e1 [[z/y]][e/x])$$

$$\text{pick } z \notin FV(e) \quad (\text{if } y \neq x \ \& \ y \in FV(e))$$

OPERATIONAL SEMANTICS

- single-step evaluation (judgment form): $e \rightarrow e'$
- primary rule (**beta reduction**):

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

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

EVALUATION STRATEGIES

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

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

$id (\underline{id (\lambda z. id z)})$

$id (id (\lambda z. \underline{id z}))$

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

$id (id (\lambda z. \underline{id z}))$

→ $id (\underline{id (\lambda z. z)})$

→ $\underline{id (\lambda z. z)}$

→ $\lambda z. z$

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 $v ::= \lambda x.e$ (lambda abstractions)

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

→ id ($\lambda z.$ id z)

→ $\lambda z.$ id z

ANOTHER EXAMPLE (DIFF BETWEEN CALL BY NAME AND CALL BY VALUE)

- Call by name:

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

→ y

- Call by value:

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

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

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

→ ...

Infinite Loop!

CALL-BY-VALUE OPERATIONAL SEMANTICS

- Basic rule

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

- Search rules:

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

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

- Notice, evaluation is left to right

ALTERNATIVES

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

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

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

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

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

call-by-value

call-by-name

ALTERNATIVES

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

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

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

call-by-value

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

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

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

normal order

ALTERNATIVES

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

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

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

call-by-value

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

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

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

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

full beta-reduction

ALTERNATIVES

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

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

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

call-by-value

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

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

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

right-to-left call-by-value

PROVING THEOREMS ABOUT O.S.

Call-by-value o.s.:

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

To prove property P of $e1 \rightarrow e2$, there are 3 cases:

case:

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

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

case:

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

IH = $P(e1 \rightarrow e1')$
 Must prove: $P(e1 e2 \rightarrow e1' e2)$

case:

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

IH = $P(e2 \rightarrow e2')$
 Must prove: $P(v e2 \rightarrow v e2')$

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 e_1 is closed and $e_1 \rightarrow^* e_2$ then e_2 is closed.

Proof: by induction on derivation of $e_1 \rightarrow^* e_2$.

(We need to prove lemma: if e_1 is closed and $e_1 \rightarrow e_2$, then e_2 is closed.)