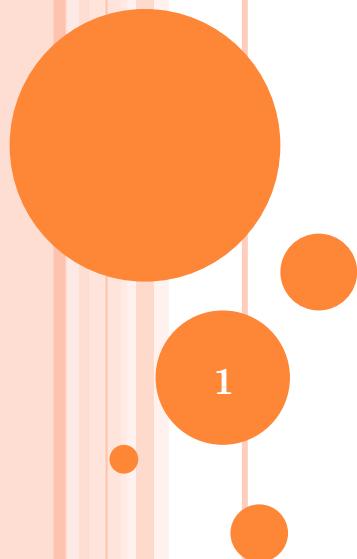


# UNTYPED LAMBDA CALCULUS



# Original $\lambda$ -CALCULUS SYNTAX

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

e ::= x	(a variable)
\x.e	(a nameless function/ <i>lambda abstraction</i> )
e <sub>1</sub> e <sub>2</sub>	(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 “ $\lambda$ ” 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)
\x.e	(a nameless function/ <b>lambda abstraction</b> )
e <sub>1</sub> e <sub>2</sub>	(function application)

- Suppose we define a judgment form:

- $e$  term ( $e$  is a lambda 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:
    - “ $y z x$ ” 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* a 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

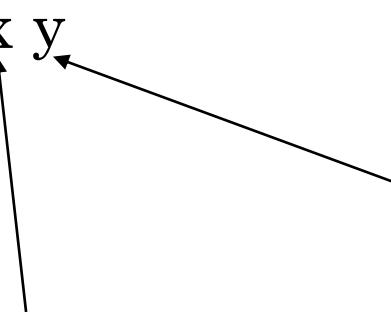
- A name and the object it denotes are NOT the same thing!
- A name is merely a “*character string*”.
- An object can have multiple names – “*aliasing*”.
- A name can denote different objects at different times.
- “variable *bar*” means “the variable with the name *bar*”.
- “function *foo*” means “the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

- $\lambda x.e$  

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)
- $\lambda x.x y$  

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

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

# FREE VARIABLES

- $\text{free}(x) = \{x\}$
- $\text{free}(e_1 e_2) = \text{free}(e_1) \sqcup \text{free}(e_2)$
- $\text{free}(\lambda x.e) = \text{free}(e) - \{x\}$

Judgement form?

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

# FREE VARIABLES (INFERENCE RULES)

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

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

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

## QUIZ: FREE VARIABLE

- Is there a free variable in the following lamda expression? If so, circle it:

$(\lambda w.w\ w)$

# ALL VARIABLES

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

$$\text{Vars}(e_1 e_2) = \text{Vars}(e_1) \cup \text{Vars}(e_2)$$

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

# 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*.

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

↑  
Alpha-renaming

Capturing!

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  $\text{FV}(e) \cap \text{Vars}(e1) = \emptyset$ :

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

# 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.

e.g.: to alpha-rename  $\lambda x.e$  we:

1. pick  $z$  such that  $z$  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)

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

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

$$e_1 e_2 [e/x] = (e_1 [e/x]) (e_2 [e/x])$$

$$(\lambda x. e_1) [e/x] = \lambda x. e_1$$

$$(\lambda y. e_1) [e/x] = \lambda y. (e_1 [e/x]) \quad (\text{if } y \neq x \text{ \& } y \notin FV(e))$$

$$= \lambda z. (e_1 [[z/y]] [e/x])$$

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

## QUIZ: SUBSTITUTION

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

# OPERATIONAL SEMANTICS

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

$$\frac{}{(\lambda x.e_1) \ e_2 \rightarrow e_1 [e_2/x]}$$

- A term of the form  $(\lambda x.e_1) \ e_2$  is called **redex** (**reducible expression**).

# EVALUATION STRATEGIES

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

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

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

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

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

- Full beta-reduction*: any redex

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

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

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

$\rightarrow \lambda z. z$

# EVALUATION STRATEGIES

- *Normal order*: leftmost, outermost redex first

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

→ id ( $\lambda z. id z$ )

→  $\lambda z. \underline{id z}$

→  $\lambda z. z$

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

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

→ id ( $\lambda z. id z$ )

→  $\lambda z. \underline{id z}$

# 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)

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

$\rightarrow \text{id } (\lambda z. \text{id } z)$

$\rightarrow \lambda z. \text{id } z$

# 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

- Judgment form:  $e \rightarrow e'$

- Basic rule

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

- Search rules:

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v e_2 \rightarrow v e_2'}$$

- Notice, evaluation is left to right (keep evaluating  $e_1$  until it becomes a **value**  $v$ )

v is a value!

# ALTERNATIVES

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

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v e_2 \rightarrow v e_2'}$$

call-by-value

$$(\lambda x.e_1) e_2 \rightarrow e_1 [e_2/x]$$

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

call-by-name

# ALTERNATIVES

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

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v e_2 \rightarrow v e_2'}$$

call-by-value

$$(\lambda x.e_1) e_2 \rightarrow e_1 [e_2/x]$$

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

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

normal order

# ALTERNATIVES

Note if multiple rules can fire at the same time, which one gets fired is non-deterministic

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

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v e_2 \rightarrow v e_2'}$$

call-by-value

$$\frac{}{(\lambda x.e_1) e_2 \rightarrow e_1 [e_2/x]}$$

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$\frac{e_2 \rightarrow e_2'}{e_1 e_2 \rightarrow e_1 e_2'}$$

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

full beta-reduction

# ALTERNATIVES

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

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v e_2 \rightarrow v e_2'}$$

call-by-value

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

$$\frac{e_1 \rightarrow e_1'}{e_1 v \rightarrow e_1' v}$$

$$\frac{e_2 \rightarrow e_2'}{e_1 e_2 \rightarrow e_1 e_2'}$$

right-to-left call-by-value

# PROVING THEOREMS ABOUT O.S.

Call-by-value o.s.:

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

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v e_2 \rightarrow v e_2'}$$

To prove property P of  $e_1 \rightarrow e_2$ , 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:

$$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$$

$$IH = P(e_1 \rightarrow e_1')$$

Must prove:  $P(e_1 e_2 \rightarrow e_1' e_2)$

case:

$$\frac{e_2 \rightarrow e_2'}{v e_2 \rightarrow v e_2'}$$

$$IH = P(e_2 \rightarrow e_2')$$

Must prove:  $P(v e_2 \rightarrow v e_2')$

# 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:”

$$\frac{}{e1 \rightarrow^* e1} \text{ (reflexivity)}$$

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

# PROVING THEOREMS ABOUT O.S.

Call-by-value o.s.:

$$\frac{}{e1 \rightarrow^* e1} \quad (\text{reflexivity})$$

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

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

case: \_\_\_\_\_

$$e1 \rightarrow^* e1$$

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

case:

$$\frac{e1 \rightarrow e2 \ e2 \rightarrow^* e3}{e1 \rightarrow^* e3}$$

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

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

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

## EXAMPLE

Definition: An expression  $e$  is **closed** if  $\text{FV}(e) = \{\}$ .

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.)