

Semantics

- To evaluate $(\lambda x.e1) e2$
 - Bind x to $e2$
 - Evaluate $e1$
 - Return the result of the evaluation
- This is called “beta reduction”
 - $(\lambda x.e1) e2 \rightarrow_{\beta} e1[e2/x]$
 - $(\lambda x.e1) e2$ is called a *redex*
 - We'll usually omit the beta

CMSC 631

7

Three Conveniences

- Syntactic sugar for local declarations
 - $\text{let } x = e1 \text{ in } e2$ is short for $(\lambda x.e2) e1$
- Scope of λ extends as far to the right as possible
 - $\lambda x.\lambda y.x y$ is $\lambda x.(\lambda y.(x y))$
- Function application is left associative
 - $x y z$ is $(x y) z$

CMSC 631

8

Scoping and Parameter Passing

- Beta reduction is not yet precise
 - $(\lambda x.e1) e2 \rightarrow e1[e2/x]$
 - what if there are multiple x 's?
- Example:
 - $\text{let } x = a \text{ in}$
 - $\text{let } y = \lambda z.x \text{ in}$
 - $\text{let } x = b \text{ in } y x$
 - which x 's are bound to a , and which to b ?

CMSC 631

9

Static (Lexical) Scope

- Just like most languages, a variable refers to the closest definition
- Make this precise using variable renaming
 - The term
– $\text{let } x = a \text{ in let } y = \lambda z.x \text{ in let } x = b \text{ in } y x$
 - is “the same” as
– $\text{let } x = a \text{ in let } y = \lambda z.x \text{ in let } w = b \text{ in } y w$
 - Variable names don't matter

CMSC 631

10

Free and Bound Variables

- The set of *free variables* of a term is
 - $FV(x) = \{x\}$
 - $FV(\lambda x.e) = FV(e) - \{x\}$
 - $FV(e1 e2) = FV(e1) \cup FV(e2)$
- A term e is *closed* if $FV(e) = \emptyset$
- A variable that is not free is *bound*

CMSC 631

11

Alpha Conversion

- Terms are equivalent up to renaming of bound variables
 - $\lambda x.e = \lambda y.(e[y/x])$ if $y \notin FV(e)$
- This is often called *alpha conversion*, and we will use it implicitly whenever we need to avoid capturing variables when we perform substitution

CMSC 631

12

Substitution

Formal definition:

- $x[e/x] = e$
- $z[e/x] = z$ if $z \neq x$
- $(e_1 e_2)[e/x] = (e_1[e/x] e_2[e/x])$
- $(\lambda z.e_1)[e/x] = \lambda z.(e_1[e/x])$ if $z \neq x$ and $z \in FV(e)$

Example:

- $(\lambda x.y x) x =_{\alpha} (\lambda w.y w) x \rightarrow_{\beta} y x$
- (We won't write alpha conversion explicitly in general)

CMSC 631

13

A Note on Substitutions

People write substitution many different ways

- $e[e_2/x]$
- $e[x \mapsto e_2]$
- $[x/e_2]e$
- and more...

But they all mean the same thing

- The variable is being substituted with the term

CMSC 631

14

Multi-Argument Functions

We can't (yet) write multi argument functions

- E.g., a function of two arguments $\lambda(x, y).e$

Trick: Take arguments one at a time

- $\lambda x.\lambda y.e$
- This is a function that, given argument x , returns a function that, given argument y , returns e
- $(\lambda x.\lambda y.e) a b \rightarrow (\lambda y.e[a/x]) b \rightarrow e[a/x][b/y]$

This is often called *Currying* and can be used to represent functions with any # of arguments

CMSC 631

15

Booleans

- $\text{true} = \lambda x.\lambda y.x$
- $\text{false} = \lambda x.\lambda y.y$
- $\text{if } a \text{ then } b \text{ else } c = a b c$

Example:

- $\text{if true then } b \text{ else } c \rightarrow (\lambda x.\lambda y.x) b c \rightarrow (\lambda y.b) c \rightarrow b$
- $\text{if false then } b \text{ else } c \rightarrow (\lambda x.\lambda y.y) b c \rightarrow (\lambda y.y) c \rightarrow c$

CMSC 631

16

Combinators

Any closed term is also called a *combinator*

- So *true* and *false* are both combinators

Other popular combinators

- $I = \lambda x.x$
- $S = \lambda x.\lambda y.x$
- $K = \lambda x.\lambda y.\lambda z.x z (y z)$
- Can also define calculi in terms of combinators
 - E.g., the SKI calculus
 - Turns out the SKI calculus is also Turing complete

CMSC 631

17

Pairs

- $(a, b) = \lambda x.\text{if } x \text{ then } a \text{ else } b$
- $\text{fst} = \lambda p.p \text{ true}$
- $\text{snd} = \lambda p.p \text{ false}$

Then

- $\text{fst } (a, b) \rightarrow^* a$
- $\text{snd } (a, b) \rightarrow^* b$

CMSC 631

18

Natural Numbers (Church)

- 0 = $\lambda x.\lambda y.y$
- 1 = $\lambda x.\lambda y.x\ y$
- 2 = $\lambda x.\lambda y.x(x\ y)$
- i.e., $n = \lambda x.\lambda y.<\text{apply } x \text{ n times to } y>$
- succ = $\lambda z.\lambda x.\lambda y.x(z\ x\ y)$
- iszero = $\lambda z.z\ (\lambda y.\text{false})\ \text{true}$

CMSC 631

19

Natural Numbers (Scott)

- 0 = $\lambda x.\lambda y.x$
- 1 = $\lambda x.\lambda y.y\ 0$
- 2 = $\lambda x.\lambda y.y\ 1$
- i.e., $n = \lambda x.\lambda y.y\ (n-1)$
- succ = $\lambda z.\lambda x.\lambda y.y\ z$
- pred = $\lambda z.z\ 0\ (\lambda x.x)$
- iszero = $\lambda z.z\ \text{true}\ (\lambda x.\text{false})$

CMSC 631

20

A Non-deterministic Small-Step Semantics

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

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

- Why are these semantics non-deterministic?

CMSC 631

21

Example

- We can apply reduction anywhere in a term
 - $(\lambda x.(\lambda y.y)\ x)\ ((\lambda z.w)\ x) \rightarrow \lambda x.(x\ ((\lambda z.w)\ x)) \rightarrow \lambda x.x\ w$
 - $(\lambda x.(\lambda y.y)\ x)\ ((\lambda z.w)\ x) \rightarrow \lambda x.(\lambda y.y\ x\ (w)) \rightarrow \lambda x.x\ w$
- Does the order of evaluation matter?

CMSC 631

22

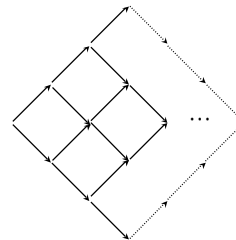
The Church-Rosser Theorem

- Lemma (The Diamond Property):
 - If $a \rightarrow b$ and $a \rightarrow c$, there exists d such that $b \rightarrow^* d$ and $c \rightarrow^* d$
- Church Rosser Theorem:
 - If $a \rightarrow^* b$ and $a \rightarrow^* c$, there exists d such that $b \rightarrow^* d$ and $c \rightarrow^* d$
- Proof: By diamond property
- Church-Rosser is also called **confluence**

CMSC 631

23

Proof



CMSC 631

24

Normal Form

- A term is in *normal form* if it cannot be reduced
 - Examples: $\lambda x.x$, $\lambda x.\lambda y.z$
 - Some normal forms referred to as values: the “legal” end results of programs
- By Church Rosser Theorem, every term reduces to at most one normal form
- Notice that for our application rule, the argument need not be a normal form

CMSC 631

25

Beta-Equivalence

- Let $=_{\beta}$ be the reflexive, **symmetric**, and transitive closure of \rightarrow
 - Usually we think only of reduction; adding symmetry extends this to equivalence
 - E.g., $(\lambda x.x) y \rightarrow y \leftarrow (\lambda z.\lambda w.z) y y$, so all three are beta equivalent
- If $a =_{\beta} b$, then $\exists c$ such that $a \rightarrow^* c$ and $b \rightarrow^* c$
 - Proof: Consequence of Church-Rosser Theorem
- In particular, if $a =_{\beta} b$ and both are normal forms, then they are equal

CMSC 631

26

Not Every Term Has a Normal Form

- Consider
 - $\Delta = \lambda x.x x$
 - Then $\Delta \Delta \rightarrow \Delta \Delta \rightarrow \dots$
- In general, *self application* leads to loops
 - ...which is good if we want recursion

CMSC 631

27

A Fixpoint Combinator

- Also called a paradoxical combinator
 - $Y = \lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))$
 - Note: There are many versions of this combinator
- Then $Y F =_{\beta} F (Y F)$ for any F
 - $Y F = (\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) F$
 - $\rightarrow (\lambda x.F (x x)) (\lambda x.F (x x))$
 - $\rightarrow F ((\lambda x.F (x x)) (\lambda x.F (x x)))$
 - $\leftarrow F (Y F)$

CMSC 631

28

Example

- Fact $n = \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}(n-1)$
- Let $G = \lambda f.<\text{body of factorial}>$
 - I.e., $G = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$
- $Y G I =_{\beta} G (Y G) I$
 - $=_{\beta} (\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) (Y G) I$
 - $=_{\beta} \text{if } I = 0 \text{ then } 1 \text{ else } I * (Y G) 0$
 - $=_{\beta} \text{if } I = 0 \text{ then } 1 \text{ else } I * (G (Y G) 0)$
 - $=_{\beta} \text{if } I = 0 \text{ then } 1 \text{ else } I * (\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) (Y G) 0$
 - $=_{\beta} \text{if } I = 0 \text{ then } 1 \text{ else } I * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * (Y G) 0)$
 - $=_{\beta} I * I = I$

CMSC 631

29

In Other Words

- The Y combinator “unrolls” or “unfolds” its argument an infinite number of times
 - $Y G = G (Y G) = G (G (Y G)) = G (G (G (Y G))) = \dots$
 - G needs to have a “base case” to ensure termination
- We can use this trick to encode arbitrary recursion
- Notice that this only works because we’re call-by-name

CMSC 631

30

Encodings

- Encodings are fun
- They show language expressiveness

- In practice, we usually add constructs as primitives
 - Much more efficient
 - Much easier to perform program analysis on and avoid silly mistakes with
 - E.g., our encodings of `true` and `0` are exactly the same, but we may want to forbid mixing booleans and integers

CMSC 631

31

Lazy vs. Eager Evaluation

- Our non-deterministic reduction rule is fine for theory, but awkward to implement

- Two deterministic strategies:
 - *Lazy*: Given $(\lambda x.e1) e2$, do not evaluate $e2$ if x does not “need” $e1$
 - Also called left-most, call-by-name, call-by-need, applicative, normal-order (with slightly different meanings)
 - *Eager*: Given $(\lambda x.e1) e2$, always evaluate $e2$ fully before applying the function
 - Also called call-by-value

CMSC 631

32

Lazy Operational Semantics

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

- The rules are deterministic
- The rules do not reduce under λ
- The rules are normalizing:
 - If a is closed and there is a normal form b such that $a \rightarrow^* b$, then $a \rightarrow^l d$ for some d

CMSC 631

33

Eager Operational Semantics

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

- This big-step semantics is also deterministic and does not reduce under λ
- But it is not normalizing
 - Example: `let x = Δ Δ in $(\lambda y.y)$`

CMSC 631

34

Lazy vs. Eager in Practice

- Lazy evaluation (call by name, call by need)
 - Has some nice theoretical properties
 - Terminates more often
 - Lets you play some tricks with “infinite” objects
 - Main example: Haskell

- Eager evaluation (call by value)
 - Is generally easier to implement efficiently
 - Blends more easily with side effects
 - Main examples: Most languages (C, Java, ML, etc.)

CMSC 631

35

Functional Programming

- The λ calculus is a prototypical functional programming language:
 - Lots of higher-order functions
 - No side-effects

- In practice, many functional programming languages are “impure” and permit side-effects
 - But you’re supposed to avoid using them

CMSC 631

36

Functional Programming Today

- Two main camps:
 - Haskell – Pure, lazy functional language; no side effects
 - ML (SML/NJ, OCaml) – Call-by-value, with side effects
- Still around: LISP, Scheme
 - Disadvantage/advantage: No static type systems

CMSC 631

37

Call-by-Name Example

```
OCaml
let cond p x y = if p then x else y
let rec loop () = loop ()
let z = cond true 42 (loop ())
```

infinite loop at call

```
Haskell
cond p x y = if p then x else y
loop () = loop ()
z = cond True 42 (loop ())
```

3rd argument never used by cond, so never invoked

CMSC 631

38

Two Cool Things to Do with CBN

- Build control structures with functions

```
cond p x y = if p then x else y
```

- “Infinite” data structures

```
integers n = n: (integers (n+1))
take 10 (integers 0) (* infinite loop in cbv *)
```

CMSC 631

39