Alternative evaluation schemes

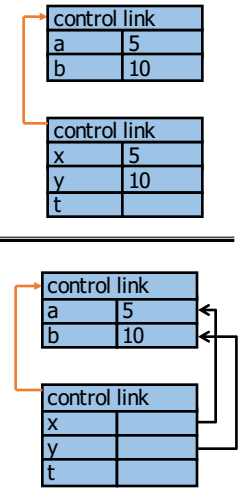
- Eager evaluation and parameter passing: by value vs. by reference
- Lambda Calculus
- Substitution
- Evaluation order
- Pass-by-name, call-by-name
- Delayed evaluation
- Laziness, call-by-need

Parameter Passing Modes

for eager evaluation

```
void swap(int x, int y) {
  int t = x;
  x = y;
  y = t;
}

void main() {
  int a = 5;
  int b = 10;
  swap(a,b);
  print a;
}
```



Pass By Value

Pass By Reference

Parameter Passing in ML

"By Ref"

```
fun swap(x, y) =
  let val t = !x in
    x := !y; y := t
  end

val a = ref 1
val b = ref 2
val (c,d) = swap(a,b)

swap(!a, !b);  <= type error
```

"By Val"

```
fun add(x, y) =
  x + y

val a = ref 1
val b = ref 2
add(!a, !a)

add(a, b);  <= type error
```

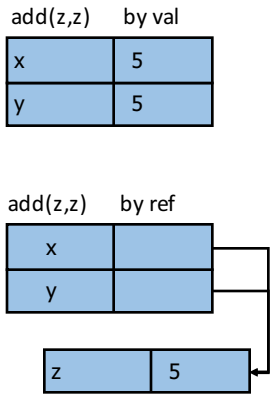
Why Does it Matter?

Usual culprit: mutation

Aliasing

```
int add(x, y) {
  x = x + 1;
  return x + y;
}
z = 5;
print add(z, z);
```

Efficiency



(lambda) λ calculus

- "simplest possible functional programming language"
- ***Reduction*** (evaluation) by rewriting function applications to ***substitute*** arguments in place of parameters.
- Invented by Alonzo Church, 1920s/1930s
  - (of the Church-Turing Thesis and more..., Turing's advisor)
- One of 3 fundamental models of computation
- Why study?
  - Essence of functions
  - Alternative to environment-based evaluation
  - Understand other choices among alternative evaluation rules.

Pure Lambda Calculus

Terms

e ::=

x	*variables*
(e e)	*application (left associative)*
(λx.e)	*lambda expression* *(think anonymous function)*

Equivalence/Reduction Rules

$(\lambda x.e) = (\lambda y.[y/x]e)$ (α-rename)

(alpha) ***α*-equivalence, *α*-renaming**

You may rename the variable bound by a lambda if you substitute its new name for its old name in all of its uses in the lambda's body.

$(\lambda x.e_1)e_2 = [e_2/x]e_1$ (β-reduce)

(beta) ***β*-equivalence, *β*-reduction**

You may reduce a lambda application to the lambda's body if you substitute the argument for all uses uf the lambda's parameter.

$[e_1/x]e_2$ means "substitute e_1 for all occurrences of **x** in e_2" or "replace all occurrence of **x** in e_2 with e_1."

Substitution Rules

$[e_1/x]e_2$

"substitute e_1 for all occurrences of **x** in e_2"
"replace all occurrence of **x** in e_2 with e_1"

$[e/x]x = e$

$[e/x]y = y$ if $y \neq x$

$[e_1/x](e_2\ e_3) = ([e_1/x]e_2)\ ([e/x]e_3))$

Take care!

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

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

if $y \neq x$ and $y \notin FV(e_1)$

Double-check your printed copy

Careful substitution

Must be **careful** with substitution to avoid ***variable capture,*** *i.e.,* accidentally attaching variable uses to wrong definitions.

Free Variables

Variables used but not bound in a term.

FV(e) is defined recursively for lambda calculus terms ***e***:

$$FV(x) = \{x\}$$
$$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$$
$$FV(\lambda x.e) = FV(e) - \{x\}$$

Why so careful?

Illegal reduction: captures z

(λf. λz. f (f z))(λx. x + z) → (λz.(λx. x + z)((λx. x + z) z)
→ λz. (λx. x + z)(z + z)
→ λz. (z + z) + z

FV( ) = {z}

Legal reduction: α-rename z to y to avoid capture.

(λf. λz. f (f z))(λx. x + z) → (λf. λy. f (f y))(λx. x + z)
→ (λy. (λx. x + z)((λx. x + z) y)
→ λy. (λx. x + z)(y + z)
→ λy.(y + z) + z

η-reduction

nice, but optional, rule

$$(\lambda x.e\ x) = e$$

Example reduction

Left-associative

```
(λf.λx.f(f  x))(λz.x+z)  2
(λf.λa.f(f  a))(λz.x+z)  2      (rename  x)
([(λz.x+z)/f]λa.f(f  a))  2     (reduce  f)
(λa.(λz.x+z)((λz.x+z)a))  2     (substitution)
(λa.(λb.x+b)((λz.x+z)a))  2     (rename  z)
(λa.(λb.x+b)([a/z](x+z))  2     (reduce  z)
(λa.(λb.x+b)(x+a))  2           (substitution)
(λa.[(x+a)/b](x+b))  2          (reduce  b)
(λa.x+(x+a))  2                 (substitution)
[2/a](x+(x+a))                  (reduce  a)
x+x+2                           (substitution)
```

Normal form: *no more reductions possible.*

12

Normal forms

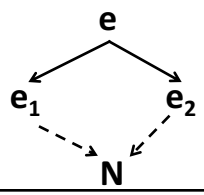
Term is in ***normal form*** if no more reductions are possible.

Not all terms can be reduced to a normal form:

Ω = (λx. (x x)) (λx. (x x))

Confluence:

If ***e*** can be reduced to a normal form, it can be reduced to exactly one normal form. Reduction order does not matter.



BUT, not all reduction orders/ evaluation strategies are guaranteed to reach a normal form.

13

Evaluation/reduction strategies

Any order allowed by the rules is valid:

(λx. x + 7)**((λy. y * 4) 2)** → (λx. x + 7) 8 → 8 + 7 → 15

(λx. x + 7)((λy. y * 4) 2) → **((λy. y * 4) 2)** + 7 → 8 + 7 → 15

Some evaluation orders do not lead to normal form.

(λx. 251) (λx.(x x)) (λx.(x x))

Encodings: booleans, conditionals

true = λt. λf. t

Curried 2-argument function returns 1st arg

false = λt. λf. f

Curried 2-argument function returns 2nd arg

if = λc. λt. λf. c t f

Curried 3-argument function applies 1st arg (boolean condition) to:

- 2nd argument (true branch) and
- 3rd argument (false branch)

Encodings: natural numbers

(Church numerals)

Numbers *n* are functions that compose their first argument (s for successor) *n* times over their second argument (z for zero).

0 = λs. λz. z

1 = λs. λz. s z

2 = λs. λz. s (s z)

…

n = λs. λz. s (s (s … (s z) …))

Encodings: increasing arithmetic

(Church numerals)

Rad!!

succ = λn. λs. λz. s (n s z)
Succ is (*n+1*)-fold composition of s over z..

add = λ m. λ n. λ s. λz. n s (m s z)
Add is (*m+n*)-fold composition of s over z..

mult = λ m. λ n. (n (add m) **0**)
Mult is *n*-fold composition of *add m* over 0.

zero? = λ n. n (λx. **false**) **true**
Zero? Is *n*-fold composition of (λx. **false**) over true.

Subtraction is tricky: represent n as pair *(n-1, n),* define **pred** function...

Recursion with fixed points (1)

and anonymous functions only

Rad!!

Open recursion: **F** = λf. λn. **if** (**zero?** n) 1 (**mult** n (f (**pred** n)))
Want to apply **F** to underlined part, to bind **f** for recursive use...

Find **fact** that is a ***fixed point*** of **F**, *i.e.*, where **fact = F fact**.
Argument value for which function returns its argument.

Y, a fixed-point combinator: **λf. (λx. f (x x)) (λx. f (x x))**
Y g = (λf. (λx. f (x x)) (λx. f (x x))) g

Y g = g (Y g) = g (g (Y g)) = g (g (g (Y g))) = ...

Recursion with fixed points (2)

and anonymous functions only

Rad!!

F = λf. λn. **if** (**zero?** n) 1 (**mult** n (f (**pred** n)))

Let **fact = Y F**. **Y F = F (Y F)** → **fact = F fact**

Legal to expand now, but don't!

fact 1 = (**F** (**fact**)) 1
= (λn. **if** (**zero?** n) 1 (**mult** n (**fact** (**pred** n)))) 1
= **if** (**zero?** 1) 1 (**mult** 1 (**fact** (**pred** 1)))
= **mult** 1 (**fact** (**pred** 1))
= **fact** 0
= (λn. **if** (**zero?** n) 1 (**mult** n (**fact** (**pred** n)))) 0
= **if** (**zero?** 0) 1 (**mult** 0 (**fact** (**pred** 0)))
= 1