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;
}

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
Alasing

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

Efficiency
(lambda) 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

$e ::= \begin{align*}
\lambda x.e & \text{ \textit{lambda expression} (think anonymous function)} \\
x & \text{ \textit{variables}} \\
(e_1 e_2) & \text{ \textit{application} (left associative)} \\
(\lambda x.e) & \text{ \textit{lambda expression}}
\end{align*}$

Equivalence/Reduction Rules

$$(\lambda x.e) = (\lambda y.[y/x]e) \quad (\alpha\text{-rename})$$

(\textit{alpha}) $\alpha$-equivalence, $\alpha$-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.

$$(\lambda x.e_1)e_2 = [e_2/x]e_1 \quad (\beta\text{-reduce})$$

(\textit{beta}) $\beta$-equivalence, $\beta$-reduction
You may reduce a lambda application to the lambda’s body if you substitute the argument for all uses of 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 $e_1$.”

$[e/x]e = e$

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

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

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

if $y \neq x$ and $y \notin \text{fv}(e_1)$

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 $e_1$"

Take care!

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 \( e \):

\[
FV(x) = \{x\} \\
FV(e_1; e_2) = FV(e_1) \cup FV(e_2) \\
FV(\lambda x.e) = FV(e) - \{x\}
\]

\( \eta \)-reduction

nice, but optional, rule

\[
(\lambda x.e \ x) = e
\]

Why so careful?

Illegal reduction: captures \( z \)

\[
(\lambda f. \lambda x. f \ (f \ z))(\lambda x. x + z) \rightarrow (\lambda z. (\lambda x. x + z)(\lambda x. x + z) \ z) \\
(\lambda z. (\lambda x. x + z)(\lambda x. x + z) \ z) \\
(\lambda z. (z + z) + z)
\]

Legal reduction: \( \alpha \)-rename \( z \) to \( y \) to avoid capture.

\[
(\lambda f. \lambda z. f \ (f \ z))(\lambda x. x + z) \rightarrow (\lambda y. (\lambda x. x + z)(\lambda x. x + z) \ y) \\
(\lambda y. (\lambda x. x + z)(\lambda x. x + z) \ y) \\
(\lambda y. (y + z) + z)
\]

Example reduction

Left-associative

Normal form: no more reductions possible.
Normal forms

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

Not all terms can be reduced to a normal form:

\[ \Omega = (\lambda x . (x \ x)) \ (\lambda x . (x \ x)) \]

**Confluence:**

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

\[ \text{BUT, not all reduction orders/evaluation strategies are guaranteed to reach a normal form.} \]

Evaluation/reduction strategies

Any order allowed by the rules is valid:

\[
(\lambda x . x + 7)((\lambda y . y * 4) 2) \rightarrow (\lambda x . x + 7) 8 \rightarrow 8 + 7 \rightarrow 15 \\
(\lambda x . x + 7)((\lambda y . y * 4) 2) \rightarrow ((\lambda y . y * 4) 2) + 7 \rightarrow 8 + 7 \rightarrow 15
\]

Some evaluation orders do not lead to normal form.

\[(\lambda x . 251) (\lambda x . (x \ x)) (\lambda x . (x \ x))\]

Encodings: booleans, conditionals

\[
\begin{align*}
\text{true} &= \lambda t . \lambda f . t \\
\text{false} &= \lambda t . \lambda f . f \\
\text{if} &= \lambda c . \lambda t . \lambda f . c \ t \ f
\end{align*}
\]

Curried 2-argument function returns 1st arg
Curried 2-argument function returns 2nd arg
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).

\[
\begin{align*}
0 &= \lambda s . \lambda z . z \\
1 &= \lambda s . \lambda z . s \ z \\
2 &= \lambda s . \lambda z . s \ (s \ z) \\
&\ldots \\
n &= \lambda s . \lambda z . s \ (s \ (s \ z) \ldots )
\end{align*}
\]
Encodings: increasing arithmetic
(Church numerals)

\texttt{succ} = \lambda n. \lambda s. \lambda z. s \, (n \, s \, z)
Succ is \((n+1)\)-fold composition of \(s\) over \(z\).

\texttt{add} = \lambda m. \lambda n. \lambda s. \lambda z. s \, (m \, s \, z)
Add is \((m+n)\)-fold composition of \(s\) over \(z\).

\texttt{mult} = \lambda m. \lambda n. (n \, \text{add} \, m \, 0)
Mult is \(n\)-fold composition of \(\text{add} \, m\) over \(0\).

\texttt{zero?} = \lambda n. n \, (\lambda x. \text{false}) \, \text{true}
Zero? is \(n\)-fold composition of \((\lambda x. \text{false})\) over \text{true}.

Subtraction is tricky: represent \(n\) as pair \((n-1, n)\), define \texttt{pred} function...

Recursion with fixed points (1)
and anonymous functions only

Open recursion: \(F = \lambda f. \lambda n. \text{if} \, (\text{zero?} \, n) \, 1 \, (\text{mult} \, n \, (f \, (\text{pred} \, n)))\)
Want to apply \(F\) to \underline{underlined part}, to bind \(f\) for recursive use...

Find \texttt{fact} that is a \underline{fixed point} of \(F\), \(i.e.,\) where \texttt{fact} = \(F \, \text{fact}\)
Argument value for which function returns its argument.

\(Y\), a fixed-point combinator: \(\lambda f. (\lambda x. f \, (x \, x)) \, (\lambda x. f \, (x \, x))\)
\(Y \, g = (\lambda f. (\lambda x. f \, (x \, x)) \, (\lambda x. f \, (x \, x))) \, g\)

\(Y \, g \, g \, (Y \, g) = g \, (g \, (Y \, g)) = g \, (g \, (g \, (Y \, g))) = \ldots\)

Recursion with fixed points (2)
and anonymous functions only

\(F = \lambda f. \lambda n. \text{if} \, (\text{zero?} \, n) \, 1 \, (\text{mult} \, n \, (f \, (\text{pred} \, n)))\)

Let \texttt{fact} = \(Y \, F\). \quad \(Y \, F = F \, (Y \, F) \rightarrow \text{fact} = F \, \text{fact}\)

\texttt{fact} 1 = \(F \, (\text{fact})\) 1
\hspace{1cm} = (\lambda n. \text{if} \, (\text{zero?} \, n) \, 1 \, (\text{mult} \, n \, (\text{fact} \, (\text{pred} \, n))) \, 1)
\hspace{1cm} = \text{if} \, (\text{zero?} \, 1) \, 1 \, (\text{mult} \, 1 \, (\text{fact} \, (\text{pred} \, 1)))
\hspace{1cm} = \text{mult} \, 1 \, (\text{fact} \, (\text{pred} \, 1))
\hspace{1cm} = \text{fact} \, 0
\hspace{1cm} = (\lambda n. \text{if} \, (\text{zero?} \, n) \, 1 \, (\text{mult} \, n \, (\text{fact} \, (\text{pred} \, n))) \, 0)
\hspace{1cm} = \text{if} \, (\text{zero?} \, 0) \, 1 \, (\text{mult} \, 0 \, (\text{fact} \, (\text{pred} \, 0)))
\hspace{1cm} = 1