Review

\[ e \rightarrow e' \]

\[ \lambda x. e \rightarrow \lambda x. e' \]

Previously wrote the first rule as follows:

\[ [e/v/x] = e' \]

The more concise axiom is more common

But the more verbose version fits better with how we will formally define substitution at the end of this lecture

Church-Rosser

The order in which you reduce is a “strategy”

Non-obvious fact — “Confluence” or “Church-Rosser”:

In this pure calculus,

If \( e \rightarrow^* e_1 \) and \( e \rightarrow^* e_2 \),

then there exists an \( e_3 \) such that \( e_1 \rightarrow^* e_3 \) and \( e_2 \rightarrow^* e_3 \)

“No strategy gets painted into a corner”

Useful: No rewriting via the full-reduction rules prevents you from getting an answer (Wow!)

Any rewriting system with this property is said to, “have the Church-Rosser property”
Equivalence via rewriting

We can add two more rewriting rules:

▶ Replace \( \lambda x. e \) with \( \lambda y. e' \) where \( e' \) is \( e \) with “free” \( x \)
  replaced with \( y \) (assuming \( y \) not already used in \( e \))

\[
\lambda x. e \rightarrow \lambda y. e[y/x]
\]

▶ Replace \( \lambda x. e \ x \) with \( e \) if \( x \) does not occur “free” in \( e \)

\[
\text{if } x \text{ is not free in } e \\
\lambda x. e \ x \rightarrow e
\]

Analogies: if \( e \) then true else false

List.map (fun x -> f x) lst

But beware side-effects/non-termination under call-by-value

No more rules to add

Now consider the system with:

▶ The 4 rules on slide 3
▶ The 2 rules on slide 5
▶ Rules can also run backwards (rewrite right-side to left-side)

Amazing: Under the natural denotational semantics (basically treat lambdas as functions), \( e \) and \( e' \) denote the same thing if and only if this rewriting system can show \( e \rightarrow^* e' \)

▶ So the rules are sound, meaning they respect the semantics
▶ So the rules are complete, meaning there is no need to add any more rules in order to show some equivalence they can’t

But program equivalence in a Turing-complete PL is undecidable

▶ So there is no perfect (always terminates, always correctly says yes or no) rewriting strategy for equivalence

Some other common semantics

We have seen “full reduction” and left-to-right CBV

▶ (OCaml is unspecified order, but actually right-to-left)

Claim: Without assignment, I/O, exceptions, . . . , you cannot distinguish left-to-right CBV from right-to-left CBV

▶ How would you prove this equivalence? (Hint: Lecture 6)

Another option: call-by-name (CBN) — even “smaller” than CBV!

\[
e \rightarrow e'
\]

\[
(\lambda x. e) e' \rightarrow e'[e'/x]
\]

\[
e_1 \rightarrow e'_1 \\
e_1 e_2 \rightarrow e'_1 e_2
\]

Diverges strictly less often than CBV, e.g., \( (\lambda y. \lambda z. z) \ e \)

Can be faster (fewer steps), but not usually (reuse args)

More on evaluation order

In “purely functional” code, evaluation order matters “only” for performance and termination

Example: Imagine CBV for conditionals!

\[
\text{let rec } f \ n = \text{if } n=0 \text{ then } 1 \text{ else } n*(f (n-1))
\]

Call-by-need or “lazy evaluation”:

▶ Evaluate the argument the first time it’s used and memoize the result
  ▶ Useful idiom for programmers too

Best of both worlds?

▶ For purely functional code, total equivalence with CBN and asymptotically no slower than CBV. (Note: asymptotic!)
▶ But hard to reason about side-effects
More on Call-By-Need

This course will mostly assume Call-By-Value

Haskell uses Call-By-Need

Example:

\[
\text{four} = \text{length} \ (9:(8+5):17:42:\text{[]});
\]

\[
\text{eight} = \text{four} + \text{four};
\]

\[
\text{main} = \text{do} \ {\text{putStrLn (show eight)});
\]

Example:

\[
\text{ones} = 1 : \text{ones};
\]

\[
\text{nats_from} \ x = x : (\text{nats_from} \ (x + 1));
\]

Zach Tatlock  CSE 505 Fall 2013, Lecture 8

Formalism not done yet
Need to define substitution (used in our function-call rule)

Shockingly subtle

Informally: \(e[e'/x]\) “replaces occurrences of \(x\) in \(e\) with \(e'\)”

Examples:

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

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

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

Substitution gone wrong:

Attempt #1:

\[
e_1[e_2/x] = e_3
\]

\[
x[e/x] = e \quad y \neq x \quad e_1[e/x] = e'_1
\]

\[
y[e/x] = y \quad (\lambda y. \ e_1)[e/x] = \lambda y. \ e'_1
\]

\[
e_1[e/x] = e'_1 \quad e_2[e/x] = e'_2
\]

\[
(e_1 \ e_2)[e/x] = e'_1 \ e'_2
\]

Recursively replace every \(x\) leaf with \(e\)

The rule for substituting into (nested) functions is wrong: If the function’s argument binds the same variable (shadowing), we should not change the function’s body

Example program: \((\lambda x. \ \lambda x. \ x) 42\)
First define the “free variables of an expression” $FV(e)$:

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

- $e_1[e_2/x] = e_3$
- $x[e/x] = e$
- $y \neq x$ 
- $y[e/x] = y$

- $\frac{e_1[e/x] = e'_1}{(\lambda y. e_1)[e/x] = \lambda y. e'_1}$
- $e'_1[e/x] = e'_2$
- $(e_1 e_2)[e/x] = e'_1 e'_2$

- $e_1[e/x] = e'_1$
- $y \neq x$
- $y \notin FV(e)$

- $(\lambda y. e_1)[e/x] = \lambda y. e'_1$

But this is a partial definition

- Could get stuck if there is no substitution

### Correct Substitution

Assume implicit systematic renaming of a binding and all its bound occurrences

- Lets one rule match any substitution into a function

And these rules:

- $e_1[e_2/x] = e_3$
- $x[e/x] = e$
- $y \neq x$
- $y[e/x] = y$

- $\frac{e_1[e/x] = e'_1}{e_1[e/\lambda y. e_1] = e'_1}$
- $e'_1[e/x] = e'_2$
- $(e_1 e_2)[e/x] = e'_1 e'_2$

- $z \neq x$
- $z \notin FV(e_1)$
- $z \notin FV(e)$

- $e_1[z/y] = e'_1$
- $e'_1[e/x] = e''_1$

- $(\lambda y. e_1)[e/x] = \lambda y. e''_1$

- You have to find an appropriate $z$, but one always exists and

- `compilerGenerated` appended to a global counter works
Some jargon

If you want to study/read PL research, some jargon for things we have studied is helpful...

- Implicit systematic renaming is $\alpha$-conversion. If renaming in $e_1$ can produce $e_2$, then $e_1$ and $e_2$ are $\alpha$-equivalent.
  - $\alpha$-equivalence is an equivalence relation

- Replacing $(\lambda x. e_1) e_2$ with $e_1[e_2/x]$, i.e., doing a function call, is a $\beta$-reduction
  - (The reverse step is meaning-preserving, but unusual)

- Replacing $\lambda x. e$ with $e$ is an $\eta$-reduction or $\eta$-contraction (since it’s always smaller)

- Replacing $e$ with $e$ with $\lambda x. e \ x$ is an $\eta$-expansion
  - It can delay evaluation of $e$ under CBV
  - It is sometimes necessary in languages (e.g., OCaml does not treat constructors as first-class functions)