Review

\[ e \rightarrow e' \]

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

\[ e_1 \rightarrow e'_1 \]

\[ e_2 \rightarrow e'_2 \]

\[ e \rightarrow e' \]

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

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

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"

Other Reduction "Strategies"

Suppose we allowed any substitution to take place in any order:

\[ e \rightarrow e' \]

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

\[ e_1 \rightarrow e'_1 \]

\[ e_2 \rightarrow e'_2 \]

\[ e \rightarrow e' \]

Programming languages do not typically do this, but it has uses:

- Optimize/pessimize/partially evaluate programs
- Prove programs equivalent by reducing them to the same term

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 \( \lambda x. e \) with \( e \) if \( x \) does not occur "free" in \( e \)

\[ x \) is not free in \( e \)

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

Analogies: \( \text{if } e \) then \( \text{true } \) else \( \text{false } \) \( \text{List.map (fun x -> i x) lst} \)

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
More on Call-By-Need

This course will mostly assume Call-By-Value

Haskell uses Call-By-Need

Example:

four = length (9:(8+5):17:42:[])  
eight = four + four

main = do { putStrLn (show eight) }

Example:

ones = 1 : ones

nats_from x = x : (nats_from (x + 1))

Formalism not done yet

Need to define substitution (used in our function-call rule)

Informally: \( e_1[x/x] \) "replaces occurrences of \( x \) in \( e \) with \( e' \)"

Examples:

\[\begin{array}{c}
\left( \lambda y. y \right) x = \left( \lambda y. y \right) x = \lambda z. z \\
\left( \lambda x. x \right) x = \left( \lambda x. x \right) x = \lambda x. x \left( \lambda x. x \right)
\end{array}\]

Substitution gone wrong

Attempt #1:

\[\begin{array}{c}
e_1[x/x] = e_3 \\
x[e/x] = e \\
y[e/x] = y \\
\lambda y. e_1[e/x] = \lambda y. e'_1 \\
e_1[e/x] = e'_1 \\
2e_2[e/x] = e_2' \\
\frac{e_1[e/x]}{(e_1 e_2)[e/x]} = e'_1 e'_2
\end{array}\]

Recursively replace every \( x \) leaf with \( e \) but respect shadowing

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

Example program: \( \left( \lambda x. \lambda x. x \right) 42 \)

Substitution gone wrong: Attempt #2

\[\begin{array}{c}
e_1[e_2/x] = e_3 \\
x[e/x] = e \\
y[e/x] = y \\
\lambda y. e_1[e/x] = \lambda y. e'_1 \\
e_1[e/x] = e'_1 \\
2e_2[e/x] = e_2' \\
\frac{e_1[e/x]}{(e_1 e_2)[e/x]} = e'_1 e'_2
\end{array}\]

Recursively replace every \( x \) leaf with \( e \) but respect shadowing

Substituting into (nested) functions is still wrong: If \( e \) uses an outer \( y \), then substitution captures \( y \) (actual technical name)

- Example program capturing \( y \): \( \left( \lambda x. \lambda y. x \right) \left( \lambda z. y \right) \rightarrow \lambda y. \left( \lambda z. y \right) \)
  - Different(!) from: \( \left( \lambda a. \lambda b. a \right) \left( \lambda z. y \right) \rightarrow \lambda b. \left( \lambda z. y \right) \)
  - Capture won’t happen under CBV/CBN if our source program has no free variables, but can happen under full reduction
Attempt #3

First define the "free variables of an expression" $FV(e)$:

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

- $e_1[e_2/x] = e_3$
- $x[e/x] = e$
- $y \neq x$ $\frac{e_1[e/x] = e'_1 \ y \neq x \ y \notin FV(e)}{\ (\lambda y. e_1)[e/x] = \lambda y. \ e'_1}$
- $y[e/x] = y$
- $e_1[e/x] = e'_1 \ e_2[e/x] = e'_2$
  $$\frac{\ (e_1 e_2)[e/x] = e'_1 e'_2}{(\lambda x. e_1)[e/x] = \lambda x. \ e'_1}$$

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:

- $e_1[e_2/x] = e_3$
- $x[e/x] = e$
- $y \neq x$ $\frac{e_1[e/x] = e'_1 \ e_2[e/x] = e'_2}{(\lambda y. e_1)[e/x] = \lambda y. \ e'_1}$
- $y[e/x] = y$
- $\frac{e_1[e/x] = e'_1 \ y \neq x \ y \notin FV(e)}{\ (\lambda y. e_1)[e/x] = \lambda y. \ e'_1}$

More explicit approach

While everyone in PL:
- Understands the capture problem
- Avoids it via implicit systematic renaming
  you may find that unsatisfying, especially if you have to implement substitution and full reduction in a meta-language that doesn’t have implicit renaming

This more explicit version also works

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

- You have to find an appropriate $z$, but one always exists and __$\text{compilerGenerated}$ 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 $\lambda x. e$ 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)

Implicit Renaming

- A partial definition because of the syntactic accident that $y$ was used as a binder
  - Choice of local names should be irrelevant/invisible
- So we allow implicit systematic renaming of a binding and all its bound occurrences
- So via renaming the rule with $y \neq x$ can always apply and we can remove the rule where $x$ is shadowed
- In general, we never distinguish terms that differ only in the names of variables (A key language-design principle!)
- So now even “different syntax trees” can be the “same term”
  - Treat particular choice of variable as a concrete-syntax thing