**Review**

- **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 by $y$ (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 $e$
    - $x$ 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

- **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”

- **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

---

**Other Reduction “Strategies”**

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

\[
\frac{e \rightarrow e'}{(\lambda x. e) e' \rightarrow e[e'/x]} \quad \frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1'e_2} \quad \frac{e_2 \rightarrow e_2'}{e_1 e_2 \rightarrow e_1 e_2'}
\]

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

- **Call-By-Value Left-To-Right Small-Step Operational Semantics:**
  - $e \rightarrow e'$
  - $(\lambda x. e) v \rightarrow e[v/x]$
  - $e_1 e_2 \rightarrow e_1'e_2$
  - $v e_2 \rightarrow v e_2'$

  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
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!

\[
\frac{e \rightarrow e'}{e \rightarrow e'}
\]

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

\[
e_1 \rightarrow e_1' \\
e_1 e_2 \rightarrow e_1' e_2
\]

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 Call-By-Need

This course will mostly assume Call-By-Value

Haskell uses Call-By-Need

Example:

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

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

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

Example:

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

\[
\text{nats} = \text{do} \ \{ \ \text{putStrLn} \ \text{show} \ x \ \}\}
\]

Substitution gone wrong

Attempt #1:

\[
\frac{x[e/x] = e_3}{y \neq x} \\
y[e/x] = y \\
(\lambda y. e_1)[e/x] = \lambda y. e_1'
\]

\[
e_1[e/x] = e_1' \\
e_2[e/x] = e_2'
\]

\[
(1 \ e_1 2)[e/x] = e_1' e_2'
\]

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 (called \textbf{variable
capture or shadowing}), we should not change the function’s body.

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

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:

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

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

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

Substitution gone wrong: Attempt #2

\[
\frac{e_1[e_2/x] = e_3}{y \neq x} \\
y[e/x] = y \\
(\lambda y. e_1)[e/x] = \lambda y. e_1'
\]

\[
e_1[e/x] = e_1' \\
e_2[e/x] = e_2'
\]

\[
(1 \ e_1 2)[e/x] = e_1' e_2'
\]

Recursively replace every \(x\) leaf with \(e\) \textbf{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 \(y\):
    \((\lambda x. \lambda y. x)(\lambda z. y) \rightarrow \lambda y. (\lambda z. y)\)
  ▶ Different(!) from: \((\lambda a. \lambda b. a)(\lambda z. y) \rightarrow \lambda b. (\lambda z. y)\)

Best of both worlds?
  ▶ For purely functional code, total equivalence with CBN and
    asymptotically no slower than CBV. (Note: \textit{asymptotic}!)
  ▶ But hard to reason about side-effects

More on evaluation order

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

Example: Imagine CBV for conditionals!

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

Call-by-need or “lazy evaluation”:
  ▶ Evaluate the argument the first time it’s used and
    \textit{memoize the result}
  ▶ Useful idiom for programmers too

Shockingly subtle
Correct Substitution

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

- Let one rule match any substitution into a function

And these rules:

\[
\begin{align*}
\frac{e_1[e_2/x] = e_3}{x[e/x] = e} \\
\frac{y \neq x \quad y[e/x] = y}{e_1[e/x] = e_1' \quad y \neq x \quad y \not\in \text{FV}(e)} \\
\frac{(\lambda y. e_1)[e/x] = \lambda y. e_1'}{(\lambda x. e_1)[e/x] = \lambda x. e_1'}
\end{align*}
\]

But this is a partial definition
- Could get stuck if there is no substitution

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

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

\[
\frac{z \neq x \quad z \not\in \text{FV}(e_1) \quad z \not\in \text{FV}(e)}{(\lambda y. e_1)[z/y] = \lambda y. e_1'}
\]

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

- Replacing \(\lambda x. e\) 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)