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 $e$)

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

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

  \[
  x \text{ is not free in } e \quad \to \quad (\lambda x. e) x \to e
  \]

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 $e \to^* e'$

- 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

Church-Rosser

The order in which you reduce is a “strategy”

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

In this pure calculus,

If $e \to^* e_1$ and $e \to^* e_2$,

then there exists an $e_3$ such that $e_1 \to^* e_3$

and $e_2 \to^* e_3$

“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:

\[
(\lambda x. e) \to \lambda y. e'[y/x]
\]

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

Review

\[
\begin{align*}
\text{\textbf{\lambda-calculus syntax:}} & \quad e ::= \lambda x. e | x | e e \\
\text{v ::= } & \lambda x. e
\end{align*}
\]

Call-By-Value Left-To-Right Small-Step Operational Semantics:

\[
\begin{align*}
e & \to e' \\
(\lambda x. e) v & \to e'[v/x] \\
e_1 & \to e_1' \\
e_2 & \to e_2'
\end{align*}
\]

Previously wrote the first rule as follows:

\[
e[v/x] = e' \\
(\lambda x. e) v \to e'
\]

- 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

\[
\begin{align*}
\text{\textbf{\lambda-calculus syntax:}} & \quad e ::= \lambda x. e | x | e e \\
\text{v ::= } & \lambda x. e
\end{align*}
\]
More on evaluation order

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

Example: Imagine CBV for conditionals!

```
let rec f n = if n=0 then 1 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

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:

\[
\begin{align*}
x[(\lambda y. y)/x] &= \lambda y. y \\
(\lambda y. y x)(\lambda z. z)/x] &= \lambda y. y \lambda z. z \\
(x x)(\lambda x. x x)/x] &= (\lambda x. x x)(\lambda x. x x)
\end{align*}
\]

Substitution gone wrong: Attempt #2

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

```
\[
\begin{align*}
x[e/x] &= e \\
y[e/x] &= y \\
(\lambda y. e_1)[e/x] &= \lambda y. e'_1 \\
e_1[e/x] &= e'_1 \\
e_2[e/x] &= e'_2 \\
(e_1 e_2)[e/x] &= e'_1 e'_2
\end{align*}
\]
```

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 $y$:
  \[
  (\lambda x. \lambda y. x) (\lambda z. y) \rightarrow \lambda y. (\lambda z. y)
  \]
  - Different(!) from: $(\lambda a. \lambda y. a) (\lambda z. y) \rightarrow \lambda b. (\lambda z. y)$
  - Capture won’t happen under CBV/CBN if our source program has no free variables, but can happen under full reduction

Substitution gone wrong

Attempt #1:

```
\[
\begin{align*}
e_1[e_2/x] &= e_3 \\
y \neq x \\
x[e/x] &= e \\
y[e/x] &= y \\
(\lambda y. e_1)[e/x] &= \lambda y. e'_1 \\
e_1[e/x] &= e'_1 \\
e_2[e/x] &= e'_2 \\
(e_1 e_2)[e/x] &= e'_1 e'_2
\end{align*}
\]
```

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 variable capture or shadowing), we should not change the function’s body.

Example program: $(\lambda x. \lambda x. x) 42$
**Attempt #3**

First define the “free variables of an expression” \( FV(e) \):

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

Let \( e_1 e_2 / x = e_3 \)

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

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:

\[
e_1[e_2/x] = e_3 \\
ex[e/x] = e \quad \frac{y \neq x}{y[e/x] = y} \quad e_1[e/x] = e'_1 \quad e_2[e/x] = e'_2 \quad (e_1 e_2)[e/x] = e'_1 e'_2 \quad (\lambda y. e_1)[e/x] = \lambda y. e'_1
\]

**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

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

- 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 \( \lambda x. e \) 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)