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

Analogies: if \( e \) then true else 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

Review

\(\lambda\)-calculus syntax:

\[
e ::= \lambda x. e \mid x \mid e \ e
\]

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

\[
e \rightarrow e'
\]

Previously wrote the first rule as follows:

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

The order in which you reduce is a “strategy”

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

In this pure calculus,

\[
\text{If } e \rightarrow^* e_1 \text{ and } e \rightarrow^* e_2, \text{ then there exists an } e_3 \text{ such that } e_1 \rightarrow^* e_3 \text{ and } e_2 \rightarrow^* e_3
\]

“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:

\[
e \rightarrow e'
\]

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

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

Church-Rosser

The order in which you reduce is a "strategy"

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

In this pure calculus,

\[
\text{If } e \rightarrow^* e_1 \text{ and } e \rightarrow^* e_2, \text{ then there exists an } e_3 \text{ such that } e_1 \rightarrow^* e_3 \text{ and } e_2 \rightarrow^* e_3
\]

“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
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!

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

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:
\[
\begin{align*}
  four & = length \ (9: (8+5):17:42:[]) \\
  eight & = four + four \\
  main & = do \{ \text{putStrLn} \ (\text{show} \ eight) \}
\end{align*}
\]

Example:
\[
\begin{align*}
  \text{ones} & = 1 : \text{ones} \\
  \text{nats_from} x & = x : (\text{nats_from} \ (x + 1))
\end{align*}
\]

Substitution gone wrong

Attempt #1:

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

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

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

Substitution gone wrong: Attempt #2

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

Recursively replace every \(x\) leaf with \(e\) \textit{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: \((\lambda a. \lambda b. a) \ (\lambda z. y) \rightarrow \lambda b. (\lambda z. y)\)
  ▶ 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” \( FV(e) \):
\[
\begin{align*}
FV(x) &= \{x\} \\
FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\lambda x. e) &= FV(e) - \{x\}
\end{align*}
\]
\[
e_1[e_2/x] = e_3
\]
\[
x[e/x] = e \\
y[e/x] = y
\]
\[
\begin{align*}
e_1[e/x] &= e'_1 & y \neq x & y \not\in FV(e) \\
(\lambda y. e_1)[e/x] &= \lambda y. e'_1
\end{align*}
\]
\[
\begin{align*}
(\lambda x. e_1)[e/x] &= \lambda x. e_1 \\
e_1[e/x] &= e'_1 & y \neq x & y \not\in FV(e) \\
(\lambda y. e_1)[e/x] &= \lambda y. e'_1
\end{align*}
\]

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

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
\[
z \neq x & z \not\in FV(e_1) & z \not\in FV(e) & e_1[z/y] = e'_1 & e_1[e/x] = e''_1 \\
(\lambda y. e_1)[e/x] &= \lambda z. e''_1
\]

- 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)