Programming Language Theory

Lambda Calculus (cont’d)
Roadmap

- Motivation for a new model (done)
- CBV lambda calculus using substitution (done (almost))
- Notes on concrete syntax (done)
- Simple Lambda encodings — LC is Turing complete! (done)
- Other reduction strategies
- Defining substitution

Further ahead:
- Types, type systems, and type safety
Review: Lambda calculus

$\lambda$-calculus syntax:

\[
e \ ::= \ \lambda x. \ e \mid x \mid e \ e \\
v \ ::= \ \lambda x. \ e
\]
Review: CBV Operational Semantics

A small-step, call-by-value (CBV), left-to-right operational semantics:

\[
e \rightarrow_{\text{cbv}} e'
\]

\[
\frac{e_1 \rightarrow_{\text{cbv}} e_1'}{e_1 e_2 \rightarrow_{\text{cbv}} e_1' e_2}
\]

\[
\frac{e_2 \rightarrow_{\text{cbv}} e_2'}{v e_2 \rightarrow_{\text{cbv}} v e_2'}
\]

\[
\frac{(\lambda x. e) v \rightarrow_{\text{cbv}} e[v/x]}{(\lambda x. e) v \rightarrow_{\text{cbv}} e'[v/x]}
\]

Previously, wrote the third rule as follows:

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

\[
(\lambda x. e) v \rightarrow_{\text{cbv}} e'
\]

- The concise axiom is more common
- The verbose rule fits better with substitution as formally defined at the end of this lecture.
Another operational semantics

Suppose we allowed substitutions to take place anywhere and in any order:

\[ e \rightarrow e' \]

\[
\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}
\]

\[
\frac{e_2 \rightarrow e'_2}{e_1 e_2 \rightarrow e_1 e'_2}
\]

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

\[
\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}
\]

Programming languages don’t 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”.
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”.

► Does IMP have the Church-Rosser property?
Church-Rosser

The order in which you reduce is a “strategy”.

Non-obvious fact (“Confluence” or “Church-Rosser”):
In this (pure) calculus,

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

“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”.

- Does IMP have the Church-Rosser property? Why?
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”.

- Does IMP have the Church-Rosser property? Why?
- Does PM have the Church-Rosser property?
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”.

- Does IMP have the Church-Rosser property? Why?
- Does PM have the Church-Rosser property? Why?
Equivalence via rewriting

We can add two more rewritings:

- (Assuming \( y \) does not occur “free” in \( e \))
  Replace \( \lambda x. \ e \) with \( \lambda y. \ e' \) where \( e' \) is \( e \) with “free” \( x \) replaced with \( y \):

\[
\frac{y \notin \text{FV}(e)}{\lambda x. \ e \rightarrow \lambda y. \ (e[y/x])}
\]

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

\[
\frac{x \notin \text{FV}(e)}{\lambda x. \ e \ x \rightarrow e}
\]

Analogies: if \( e \) then true else false \( \leftrightarrow e \)  
List.map (fn x => f x) l \( \leftrightarrow \) List.map f l

But beware side-effects/non-termination under call-by-value.
Equivalence via rewriting

Now consider the system with:

- The 4 rules from slide 5
- The 2 rules from slide 7
- And 6 “backward” rules (rewrite right-side to left-side)

Amazing Theorem: Under the accepted denotational semantics (not in PLT, but basically treat lambdas as “math” functions), $e$ and $e'$ denote the same thing if and only if $e \rightarrow^* e'$ (is derivable).

- The rules are sound: they respect the semantics
  - (the “if”-direction)

- The rules are complete: no need for more rules
  - (the “only if”-direction)

But, program equivalence in a Turing-complete PL is undecidable. So . . .
Equivalent via rewriting

Now consider the system with:

- The 4 rules from slide 5
- The 2 rules from slide 7
- And 6 “backward” rules (rewrite right-side to left-side)

Amazing Theorem: Under the accepted denotational semantics (not in PLT, but basically treat lambdas as “math” functions), $e$ and $e'$ denote the same thing if and only if $e \rightarrow^* e'$ (is derivable).

- The rules are sound: they respect the semantics
  - (the “if”-direction)
- The rules are complete: no need for more rules
  - (the “only if”-direction)

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.
Some other common semantics

A small-step, call-by-value (CBV), right-to-left operational semantics:

\[ e \xrightarrow{\text{cbv}} e' \]

\[
\frac{e_2 \xrightarrow{\text{cbv}} e'_2}{e_1 \ e_2 \xrightarrow{\text{cbv}} e'_1 \ e'_2}
\]

\[
\frac{e_1 \xrightarrow{\text{cbv}} e'_1}{e_1 \ v \xrightarrow{\text{cbv}} e'_1 \ v}
\]

\[
(\lambda x. \ e) \ v \xrightarrow{\text{cbv}} e[v/x]
\]
Some other common semantics

Large-step, call-by-value (CBV) operational semantics:

\[ e \Downarrow_{\text{cbv}} v' \]

\[
\lambda x. e \Downarrow_{\text{cbv}} \lambda x. e
\]

\[
e_1 \Downarrow_{\text{cbv}} \lambda x. e_1' \\
e_2 \Downarrow_{\text{cbv}} v_2 \\
e_1'[v_2/x] \Downarrow_{\text{cbv}} v_3
\]

\[
e_1 e_2 \Downarrow_{\text{cbv}} v_3
\]
Some other common semantics

Large-step, \textit{call-by-value (CBV)} operational semantics:

$$e \Downarrow_{\text{cbv}} v'$$

$$\frac{\lambda x. e \Downarrow_{\text{cbv}} \lambda x. e}{\lambda x. e \Downarrow_{\text{cbv}} \lambda x. e}$$

$$\frac{e_1 \Downarrow_{\text{cbv}} \lambda x. e'_1}{e_1 \Downarrow_{\text{cbv}} \lambda x. e'_1}$$

$$\frac{e_2 \Downarrow_{\text{cbv}} v_2}{e_2 \Downarrow_{\text{cbv}} v_2}$$

$$\frac{e'_1[v_2/x] \Downarrow_{\text{cbv}} v_3}{e'_1[v_2/x] \Downarrow_{\text{cbv}} v_3}$$

$$\frac{e_1 e_2 \Downarrow_{\text{cbv}} v_3}{e_1 e_2 \Downarrow_{\text{cbv}} v_3}$$

Is this semantics left-to-right or right-to-left or neither?
Some other common semantics

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

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: Both are equivalent to the large-step CBV.)
Some other common semantics

A call-by-name (CBN) operational semantics:

\[ e \rightarrow_{\text{cbn}} e' \]

\[ \frac{e_1 \rightarrow_{\text{cbn}} e'_1}{e_1 e_2 \rightarrow_{\text{cbn}} e'_1 e_2} \]

\[ (\lambda x.\ e)' \rightarrow_{\text{cbn}} e'[e'/x] \]

\[ e \Downarrow_{\text{cbn}} v' \]

\[ \frac{\lambda x.\ e \Downarrow_{\text{cbn}} \lambda x.\ e}{e_1 \Downarrow_{\text{cbn}} \lambda x.\ e_1 \quad e_1'[e_2/x] \Downarrow_{\text{cbn}} v_3} \]

\[ e_1 e_2 \Downarrow_{\text{cbn}} v_3 \]

Even “smaller” than CBV!

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 “only” matters for performance and termination.

Example: Imagine CBV for conditionals!

```haskell
fun f n = if n=0 then 1 else n*(f (n-1))
```
More on evaluation order

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

Example: Imagine CBV for conditionals!

```
fun f n = if n=0 then 1 else n*(f (n-1))
```

Call-by-need or “lazy evaluation” (e.g., Haskell):

- “Best of both worlds”?  
- Evaluate the argument the first time it is used. Memoize the result.
  - (Useful idiom for programmers, too.)
- Can be formalized, but it’s tricky.
  - A natural semantics for lazy evaluation; POPL’93; John Launchbury.
  - A call-by-need lambda calculus; POPL’95; Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler.
  - Call-by-need is clairvoyant call-by-value; ICFP’19; Jennifer Hackett and Graham Hutton.
More on evaluation order

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

Example: Imagine CBV for conditionals!

```haskell
fun f n = if n=0 then 1 else n*(f (n-1))
```

Call-by-need or “lazy evaluation” (e.g., Haskell):

- “Best of both worlds”?  
- Evaluate the argument the first time it is used. Memoize the result.  
  - (Useful idiom for programmers, too.)  
- Can be formalized, but it’s tricky.

For purely functional code, Call-by-need has total equivalence with CBN and same asymptotic time as CBV.

- (Note: *asymptotic*)

Hard to reason about if language has side-effects.
Substitution

Need to define substitution (used in function-application rules).

- Shockingly subtle

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

\[
x[(\lambda y. y)/x] = \lambda y. y \\
(\lambda y. y x)[(\lambda z. z)/x] = \lambda y. y \lambda z. z \\
(x x)[(\lambda x. x x)/x] = (\lambda x. x x)(\lambda x. x x)
\]
Substitution (Attempt 1)

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

\[
\begin{align*}
x[e/x] &= e \\
y &= x \\
y[e/x] &= y
\end{align*}
\]

\[
\begin{align*}
x[e/x] &= e \\
y[e/x] &= y
\end{align*}
\]

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

\[
\begin{align*}
e_1[e/x] &= e'_1 \\
(\lambda y \; e_1)[e/x] &= \lambda y \; e'_1
\end{align*}
\]
Substitution (Attempt 1)

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

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

What should happen with \((\lambda x. \lambda x. \ x) \ 42\)?
What actually happens with \((\lambda x. \ x)[42/x]\)?
Substitution (Attempt 1)

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

- \( x[e/x] = e \)
- \( y \neq x \)
- \( y[e/x] = y \)
- \( e_1[e/x] = e'_1 \quad e_2[e/x] = e'_2 \)
- \( (e_1 e_2)[e/x] = e'_1 e'_2 \)

\[ e_1[e/x] = e'_1 \]

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

What should happen with \((\lambda x. \lambda x. x) 42\)?

What actually happens with \((\lambda x. x)[42/x]\)?

If the function’s argument binds the same variable (shadowing), then we should not change the function’s body.

- “This isn’t the variable you’re looking for.”
Substitution (Attempt 2)

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

\[ x[e/x] = e \]
\[ y[e/x] = y \]
\[ \lambda x. \ e_1[e/x] = \lambda x. \ e_1 \]

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

\[ (\lambda y. \ e_1)[e/x] = \lambda y. \ e'_1 \]
Substitution (Attempt 2)

\[
\begin{align*}
&e_1[e_2/x] = e_3 \\
\hline
&x[e/x] = e \\
&y \neq x \\
&y[e/x] = y \\
(\lambda x. e_1)[e/x] = \lambda x. e_1 \\
&y \neq x \\
&e_1[e/x] = e_1' \\
&e_2[e/x] = e_2' \\
&(e_1 e_2)[e/x] = e_1' e_2' \\
&y \neq x \\
&e_1[e/x] = e_1' \\
&(\lambda y. e_1)[e/x] = \lambda y. e_1' \\
&(\lambda x. \lambda y. x) (\lambda z. y)? \\
&\text{What actually happens with } (\lambda x. \lambda y. x) (\lambda z. y)?
\end{align*}
\]
Substitution (Attempt 2)

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

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

What should happen with \((\lambda x. \ \lambda y. \ x) \ (\lambda z. \ y)\)?

What actually happens with \((\lambda x. \ \lambda y. \ x) \ (\lambda z. \ y)\)?

If \(y\) is free in \(e\), then we should not capture \(y\).

Doesn’t happen under CBV if source program has no free variables.
Can happen under other reduction strategies (e.g., full reduction).
Substitution (Attempt 3)

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

$$
FV(x) = \{x\}
$$

$$
FV(e_1 e_2) = FV(e_1) \cup FV(e_2)
$$

$$
FV(\lambda x. e) = FV(e) \setminus \{x\}
$$
Substitution (Attempt 3)

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

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\lambda x. e) &= FV(e) \setminus \{x\}
\end{align*}
\]

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

\[
\begin{array}{c}
x[e/x] = e \\
y[e/x] = y
\end{array}
\]

\[
\begin{array}{c}
e_1[e/x] = e'_1 \\
e_2[e/x] = e'_2
\end{array}
\]

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

\[
(\lambda x. e_1)[e/x] = \lambda x. e_1
\]

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

But a partial definition

\(\text{▶ (could get stuck because there is no substitution)}\)
Substitution (Attempt 4)

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

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

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

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

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

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

A correct definition

- but such renaming (and simplicity of \( e_1[z/x] = e_3 \)) motivates another solution
Substitution: Implicit Renaming

- Attempt 3 was a partial definition because of the syntactic accident that $y$ was used as a binder (a $\lambda$-bound variable)
  - but, choice of local names should be irrelevant/invisible
- So we allow implicit systematic renaming
  of a binding and all its bound occurrences.
- So the rule with $y \neq x$ and $y \notin FV(e)$ (fifth rule) can always apply.
  - Drop the rule for shadowing (fourth rule)
  - Drop the rule for explicit renaming (sixth rule)
- In general, we never distinguish terms
  that differ only in the names of bound variables.
  - (A key language-design principle!)
- So now even “different syntax trees” can be the “same term”.
  - Treat particular choice of variables as a concrete-syntax thing.
Substitution (Final)

Work with terms “up to renaming of bound variables” (“up to alpha-conversion”).

\[
\begin{align*}
FV(x) & = \{x\} \\
FV(e_1 e_2) & = FV(e_1) \cup FV(e_2) \\
FV(\lambda x. e) & = FV(e) \setminus \{x\}
\end{align*}
\]

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

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

\[
\begin{align*}
y & \neq x \\
y & \not\in FV(e) \\
(\lambda y. e_1)[e/x] & = \lambda y. e'_1
\end{align*}
\]

Substitution often thought of as a metafunction, not a judgement.
Substitution in CBV operational semantics

Small-step, *call-by-value (CBV)*, left-to-right operational semantics:

\[ e \rightarrow_{cbv} e' \]

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

Substitution often thought of as a metafunction, not a judgement.

- Many nondeterministic languages
- But no nondeterministic definitions of substitution
  - Remember: terms that differ in names of bound variables are equal
Substitution in CBV operational semantics

Small-step, call-by-value (CBV), left-to-right operational semantics:

\[ e \rightarrow_{\text{cbv}} e' \]

\[ e_1 \rightarrow_{\text{cbv}} e'_1 \quad e_2 \rightarrow_{\text{cbv}} e'_2 \quad v \quad e_2 \rightarrow_{\text{cbv}} v \quad e'_2 \]

\[ (\lambda x. e) \quad v \rightarrow_{\text{cbv}} e[v/x] \]

Substitution often thought of as a metafunction, not a judgement.

- Many nondeterministic languages
- But no nondeterministic definitions of substitution
  - Remember: terms that differ in names of bound variables are equal
Substitution: Explicit Renaming

Although 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 (e.g., SML).

In those situations, Substitution (Attempt 4) suggests the implementation:

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

► Must find an appropriate \( z \), but one always exists.
► Use a global counter and append \$_compilerGenerated
Technical Jargon

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

- Implicit systematic renaming is \( \alpha \)-conversion. If renaming in \( e \) can produce \( e' \), then \( e \) and \( e' \) are \( \alpha \)-equivalent.

- \( \alpha \)-equivalence is an equivalence relation

- Replacing \((\lambda x. e_1) \ e_2\) with \(e_1[e_2/x]\) (i.e., a function application) is a \( \beta \)-reduction

- (The reverse reduction is meaning-preserving, but unusual.)

- Replacing \(\lambda x. e \ x\) with \(e\) is an \( \eta \)-reduction or \( \eta \)-contraction

- Replacing \(e\) 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., Standard ML and the value restriction)
Roadmap

- Motivation for a new model (done)
- CBV lambda calculus using substitution (done)
- Notes on concrete syntax (done)
- Simple Lambda encodings — LC is Turing complete! (done)
- Other reduction strategies (done)
- Defining substitution (done)

Next steps:
- Types, type systems, and type safety