CS 4110

Programming Languages & Logics

Lecture 14
More λ-calculus

26 September 2016

Announcements

- Homework #3 returned
  - Out of 40, $\bar{x} = 35.9$ and $\sigma = 8$
- Homework #4 due Wednesday
- Preliminary Exam I **next Wednesday**, October 5
  - Topics: Up through Hoare logic. (No λ-calculus.)
  - In class; 50 minutes. (Show up on time to get all 50 minutes.)
  - Closed book and closed notes.
  - If the problems use any definitions (the operational semantics for IMP, the Hoare logic proof rules, etc.), those will be provided.
  - Practice problems now available on CMS.

Review: λ-calculus

Syntax

$$
\begin{array}{rcl}
e & ::= & x \mid e_1\, e_2 \mid \lambda x.\, e \\
v & ::= & \lambda x.\, e
\end{array}
$$

Semantics (call by value)

$$
\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e \to e'}{v\, e \to v\, e'}
$$

$$
\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}\ \beta
$$

Example: Twice

Consider the function defined by $\mathit{double}\ x = x + x$.

Example: Twice

Consider the function defined by $\mathit{double}\ x = x + x$.

Now suppose we want to apply *double* multiple times:

Example: Twice

Consider the function defined by $\mathit{double}\ x = x + x$.

Now suppose we want to apply *double* multiple times:

$$\mathit{quadruple}\ x \quad = \quad \mathit{double}\ (\mathit{double}\ x)$$

Example: Twice

Consider the function defined by $\mathit{double}\, x = x + x$.

Now suppose we want to apply double multiple times:

$$\begin{aligned} \mathit{quadruple}\, x &= \mathit{double}\, (\mathit{double}\, x) \\ \mathit{octuple}\, x &= \mathit{quadruple}\, (\mathit{quadruple}\, x) \end{aligned}$$

Example: Twice

Consider the function defined by $\mathit{double}\ x = x + x$.

Now suppose we want to apply *double* multiple times:

$$\begin{aligned}
\mathit{quadruple}\ x &= \mathit{double}\ (\mathit{double}\ x) \\
\mathit{octuple}\ x &= \mathit{quadruple}\ (\mathit{quadruple}\ x) \\
\mathit{hexadecatuple}\ x &= \mathit{octuple}\ (\mathit{octuple}\ x)
\end{aligned}$$

Example: Twice

Consider the function defined by $\mathit{double}\ x = x + x$.

Now suppose we want to apply *double* multiple times:

$$\begin{array}{rcl} \mathit{quadruple}\ x & = & \mathit{double}\ (\mathit{double}\ x) \\ \mathit{octuple}\ x & = & \mathit{quadruple}\ (\mathit{quadruple}\ x) \\ \mathit{hexadecatuple}\ x & = & \mathit{octuple}\ (\mathit{octuple}\ x) \end{array}$$

We can abstract this pattern using a generic function:

$$\mathit{twice} \triangleq \lambda f.\ \lambda x.\ f\,(f\,x)$$

Example: Twice

Consider the function defined by $\mathit{double}\; x = x + x$.

Now suppose we want to apply *double* multiple times:

$$\begin{array}{rcl} \mathit{quadruple}\; x & = & \mathit{double}\; (\mathit{double}\; x) \\ \mathit{octuple}\; x & = & \mathit{quadruple}\; (\mathit{quadruple}\; x) \\ \mathit{hexadecatuple}\; x & = & \mathit{octuple}\; (\mathit{octuple}\; x) \end{array}$$

We can abstract this pattern using a generic function:

$$\mathit{twice} \triangleq \lambda f.\, \lambda x.\, f\,(f\,x)$$

Now the functions above can be written as

$$\begin{array}{rcl} \mathit{quadruple} & = & \mathit{twice}\; \mathit{double} \\ \mathit{octuple} & = & \mathit{twice}\; \mathit{quadruple} \\ \mathit{hexadecatuple} & = & \mathit{twice}\; \mathit{octuple} \\ & & (\text{or } \mathit{twice}\; (\lambda x.\, \mathit{twice}\; x)) \end{array}$$

Evaluation

The essence of λ-calculus evaluation is the β-reduction rule, which says how to apply a function to an argument.

$$\frac{}{(\lambda x.\, e)\; v \rightarrow e\{v/x\}}\ \beta\text{-REDUCTION}$$

But there are many different evaluation strategies, each corresponding to particular ways of using β-reduction:

- Call-by-value
- Call-by-name
- “Full” β-reduction
- ...

Call by value

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$$

$$\frac{}{(\lambda x.\, e_1)\ v_2 \rightarrow e_1\{v_2/x\}}\ \beta$$

Key characteristics:

- Arguments evaluated fully before they are supplied to functions
- Evaluation goes from left to right (in this presentation)
- We don't evaluate "under a λ"

Call by name

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2}$$

$$\frac{}{(\lambda x.\, e_1)\, e_2 \to e_1\{e_2/x\}}\ \beta$$

Key characteristics:

- Arguments supplied immediately to functions
- Evaluation still goes from left to right (in this presentation)
- We still don't evaluate "under a λ"

Full β reduction

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{e_1\, e_2 \to e_1\, e_2'}$$

$$\frac{e \to e'}{\lambda x.\, e \to \lambda x.\, e'}$$

$$\frac{}{(\lambda x.\, e_1)\, e_2 \to e_1\{e_2/x\}}\ \beta$$

Key characteristics:

- Use the β rule anywhere...
- ...including "under a λ"...
- ...nondeterministically.

Confluence

Full β reduction has this property:

$$\begin{tikzcd} & e \arrow[dl] \arrow[dr] & \\ e_1 \arrow[dr] & & e_2 \arrow[dl] \\ & e' & \end{tikzcd}$$

Confluence

Full β reduction has this property:

$$\begin{array}{ccc} & e & \\ \swarrow & & \searrow \\ e_1 & & e_2 \\ \searrow & & \swarrow \\ & e' & \end{array}$$

Theorem (Confluence)

If $e \to^* e_1$ and $e \to^* e_2$ then $e_1 \to^* e'$ and $e_2 \to^* e'$ for some e'.

Substitution

The main workhorse in the β rule is substitution, which replaces free occurrences of a variable x with a term e.

However, defining substitution $e_1\{e_2/x\}$ correctly is tricky...

“Substitution”

As a first attempt, consider:

$$y\{e/x\} \quad = \quad \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

"Substitution"

As a first attempt, consider:

$$
\begin{array}{rcl}
y\{e/x\} & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(e_1\, e_2)\{e/x\} & = & (e_1\{e/x\})\,(e_2\{e/x\})
\end{array}
$$

“Substitution”

As a first attempt, consider:

$$
\begin{aligned}
y\{e/x\} &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(e_1\, e_2)\{e/x\} &= (e_1\{e/x\})\,(e_2\{e/x\}) \\
(\lambda y.e_1)\{e/x\} &= \lambda y.e_1\{e/x\}
\end{aligned}
$$

“Substitution”

As a first attempt, consider:

$$
\begin{array}{rcl}
y\{e/x\} & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(e_1\, e_2)\{e/x\} & = & (e_1\{e/x\})\,(e_2\{e/x\}) \\
(\lambda y.e_1)\{e/x\} & = & \lambda y.e_1\{e/x\}
\end{array}
$$

What’s wrong with this definition?

“Substitution”

As a first attempt, consider:

$$
\begin{aligned}
y\{e/x\} &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(e_1\, e_2)\{e/x\} &= (e_1\{e/x\})\,(e_2\{e/x\}) \\
(\lambda y.e_1)\{e/x\} &= \lambda y.e_1\{e/x\}
\end{aligned}
$$

What’s wrong with this definition?

It substitutes bound variables too!

$$(\lambda y.y)\{3/y\}$$

“Substitution”

As a first attempt, consider:

$$\begin{array}{rcl} y\{e/x\} & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\ (e_1\ e_2)\{e/x\} & = & (e_1\{e/x\})\ (e_2\{e/x\}) \\ (\lambda y.e_1)\{e/x\} & = & \lambda y.e_1\{e/x\} \end{array}$$

What’s wrong with this definition?

It substitutes bound variables too!

$$(\lambda y.y)\{3/y\} = (\lambda y.3)$$

““Substitution””

Okay… let’s avoid rewriting bound variables by relying on α-equivalence. We’ll require that abstractions don’t use *x*, the variable we’re substituting.

““Substitution””

Okay... let’s avoid rewriting bound variables by relying on α-equivalence. We’ll require that abstractions don’t use x, the variable we’re substituting.

$$
\begin{aligned}
y\{e/x\} &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(e_1\, e_2)\{e/x\} &= (e_1\{e/x\})\,(e_2\{e/x\})
\end{aligned}
$$

““Substitution””

Okay... let’s avoid rewriting bound variables by relying on α-equivalence. We’ll require that abstractions don’t use *x*, the variable we’re substituting.

$$\begin{array}{rcll} y\{e/x\} & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} & \\ (e_1\, e_2)\{e/x\} & = & (e_1\{e/x\})\,(e_2\{e/x\}) & \\ (\lambda y.e_1)\{e/x\} & = & \lambda y.e_1\{e/x\} & \text{where } y \neq x \end{array}$$

We *assume away* abstractions over *x*. (Thanks, α-equivalence!)

““Substitution””

Okay... let’s avoid rewriting bound variables by relying on α-equivalence. We’ll require that abstractions don’t use *x*, the variable we’re substituting.

$$
\begin{array}{rcll}
y\{e/x\} & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} & \\
(e_1\, e_2)\{e/x\} & = & (e_1\{e/x\})\,(e_2\{e/x\}) & \\
(\lambda y.e_1)\{e/x\} & = & \lambda y.e_1\{e/x\} & \text{where } y \neq x
\end{array}
$$

We *assume away* abstractions over *x*. (Thanks, α-equivalence!)

What’s wrong with this definition?

““Substitution””

Okay... let’s avoid rewriting bound variables by relying on α-equivalence. We’ll require that abstractions don’t use *x*, the variable we’re substituting.

$$
\begin{aligned}
y\{e/x\} &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(e_1\, e_2)\{e/x\} &= (e_1\{e/x\})\,(e_2\{e/x\}) \\
(\lambda y.e_1)\{e/x\} &= \lambda y.e_1\{e/x\} \qquad\qquad \text{where } y \neq x
\end{aligned}
$$

We *assume away* abstractions over *x*. (Thanks, α-equivalence!)

What’s wrong with this definition?

It leads to variable capture!

$$(\lambda y.x)\{y/x\}$$

““Substitution””

Okay... let’s avoid rewriting bound variables by relying on α-equivalence. We’ll require that abstractions don’t use x, the variable we’re substituting.

$$\begin{aligned} y\{e/x\} &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\ (e_1\, e_2)\{e/x\} &= (e_1\{e/x\})\,(e_2\{e/x\}) \\ (\lambda y.e_1)\{e/x\} &= \lambda y.e_1\{e/x\} \qquad\qquad \text{where } y \neq x \end{aligned}$$

We *assume away* abstractions over x. (Thanks, α-equivalence!)

What’s wrong with this definition?

It leads to variable capture!

$$(\lambda y.x)\{y/x\} = (\lambda y.y)$$

Real Substitution

The correct definition is *capture-avoiding substitution:*

$$
\begin{array}{rcll}
y\{e/x\} & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} & \\
(e_1\ e_2)\{e/x\} & = & (e_1\{e/x\})\ (e_2\{e/x\}) & \\
(\lambda y.e_1)\{e/x\} & = & \lambda y.(e_1\{e/x\}) & \text{where } y \neq x \text{ and } y \notin fv(e)
\end{array}
$$

where $fv(e)$ is the *free variables* of a term e.