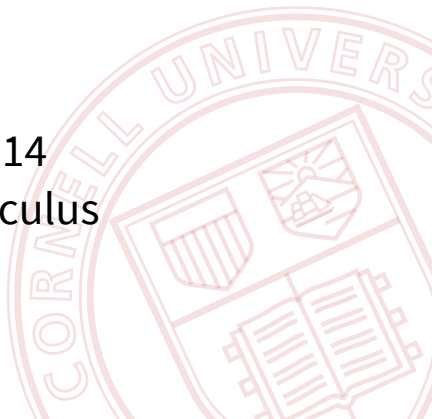


CS 4110

Programming Languages & Logics

Lecture 14
More λ -calculus



Review: λ -calculus

Syntax

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

Semantics (call by value)

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{e \rightarrow e'}{v e \rightarrow v e'}$$

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

Example: Twice

Consider the function defined by *double* $x = x + x$.

Example: Twice

Consider the function defined by *double* $x = x + x$.

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

Example: Twice

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

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

$$\textit{quadruple } x = \textit{double } (\textit{double } x)$$

Example: Twice

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

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

$$\text{quadruple } x = \text{double}(\text{double } x)$$

$$\text{hexadecatuple } x = \text{quadruple}(\text{quadruple } x)$$

Example: Twice

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

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

$$\begin{aligned}\textit{quadruple } x &= \textit{double } (\textit{double } x) \\ \textit{hexadecatuple } x &= \textit{quadruple } (\textit{quadruple } x) \\ \textit{256uple } x &= \textit{hexadecatuple } (\textit{hexadecatuple } x)\end{aligned}$$

Example: Twice

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

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

$$\begin{aligned} quadruple\ x &= double\ (double\ x) \\ hexadecatuple\ x &= quadruple\ (quadruple\ x) \\ 256uple\ x &= hexadecatuple\ (hexadecatuple\ x) \end{aligned}$$

We can abstract this pattern using a generic function:

$$twice \triangleq \lambda f. \lambda x. f(f\ x)$$

Example: Twice

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

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

$$\begin{aligned} quadruple\ x &= double\ (double\ x) \\ hexadecatuple\ x &= quadruple\ (quadruple\ x) \\ 256uple\ x &= hexadecatuple\ (hexadecatuple\ x) \end{aligned}$$

We can abstract this pattern using a generic function:

$$twice \triangleq \lambda f. \lambda x. f(f\ x)$$

Now the functions above can be written as

$$\begin{aligned} quadruple &= twice\ double \\ hexadecatuple &= twice\ quadruple \\ 256uple &= twice\ hexadecatuple \\ &\quad (or\ (twice\ (\lambda x. twice\ x))\ double) \end{aligned}$$

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

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

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

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

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \qquad \frac{e_2 \rightarrow e'_2}{e_1 e_2 \rightarrow e_1 e'_2}$$

$$\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}$$

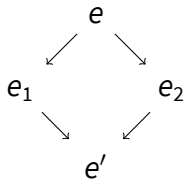
$$\frac{}{(\lambda x. e_1) e_2 \rightarrow e_1 \{e_2/x\}} \beta$$

Key characteristics:

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

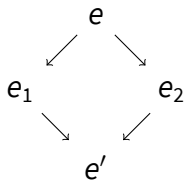
Confluence

Full β reduction has this property:



Confluence

Full β reduction has this property:



Theorem (Confluence)

If $e \rightarrow^ e_1$ and $e \rightarrow^* e_2$ then $e_1 \rightarrow^* e'$ and $e_2 \rightarrow^* e'$ 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:

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

“Substitution”

As a first attempt, consider:

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

“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:

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

What's wrong with this definition?

“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:

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

What's wrong with this definition?

It substitutes bound variables too!

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

““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.

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

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.

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

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.

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

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.

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

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*:

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

where $\text{fv}(e)$ is the *free variables* of a term e .