# CS 4110

# Programming Languages & Logics

Lecture 14
More $\lambda$-calculus

# Review: $\lambda$-calculus

## Syntax

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

## Semantics (call by value)

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

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

# Example: Twice

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

# Example: Twice

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

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:

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

# Example: Twice

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

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

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

# Example: Twice

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

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

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

# Example: Twice

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

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

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

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:

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

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

$$\begin{array}{rcl} \mathit{quadruple} & = & \mathit{twice}\ \mathit{double} \\ \mathit{hexadecatuple} & = & \mathit{twice}\ \mathit{quadruple} \\ \mathit{256uple} & = & \mathit{twice}\ \mathit{hexadecatuple} \\ & & (\text{or } (\mathit{twice}\ (\lambda x.\ \mathit{twice}\ x))\ \mathit{double}) \end{array}$$

# Evaluation

The essence of $\lambda$-calculus evaluation is the $\beta$-reduction rule, which says how to apply a function to an argument.

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

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

- Call-by-value
- Call-by-name
- “Full” $\beta$-reduction
- ...

# Call by value

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{v_1\, e_2 \to v_1\, e_2'}$$

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

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 $\lambda$"

# 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 $\lambda$"

# Full $\beta$ 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 $\beta$ rule anywhere...
- ...including "under a $\lambda$"...
- ...nondeterministically.

# Confluence

Full $\beta$ reduction has this property:

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

# Confluence

Full $\beta$ reduction has this property:

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

**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 $\beta$ 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:

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

# “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:

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

What's wrong with this definition?

It substitutes bound variables too!

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

# “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 $\alpha$-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 $\alpha$-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 $\alpha$-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, $\alpha$-equivalence!)

# ““Substitution””

Okay... let’s avoid rewriting bound variables by relying on $\alpha$-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, $\alpha$-equivalence!)

What’s wrong with this definition?

# ““Substitution””

Okay... let’s avoid rewriting bound variables by relying on $\alpha$-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, $\alpha$-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 $\alpha$-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, $\alpha$-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 $e$.