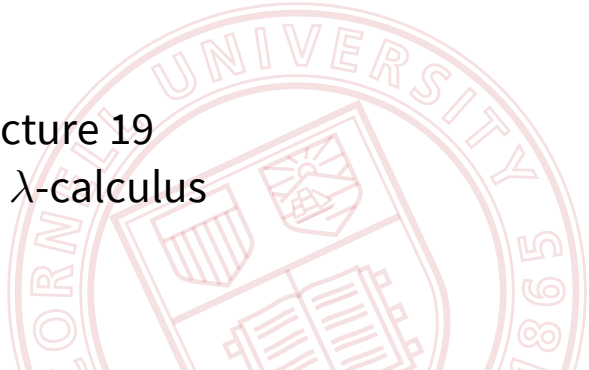
# CS 4110

# Programming Languages & Logics



Lecture 19

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)

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

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

# Example: Twice

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

# 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 $\mathit{double}\, x = x + x$.

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:

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

# Example: Twice

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

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

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

# Example: Twice

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

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

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

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 $double\ x = x + x$.

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

$$\begin{aligned} quadruple &= twice\ double \\ hexadecatuple &= twice\ quadruple \\ 256uple &= twice\ hexadecatuple \end{aligned}$$

# 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

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

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

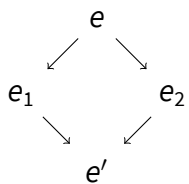
$$\frac{}{(\lambda x.\ e_1)\ e_2 \to e_1\{e_2/x\}}\ \beta$$

Key characteristics:

- Use the $\beta$ rule anywhere...
- ...including "under a $\lambda$"...
- ...nondeterministically

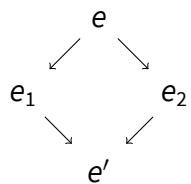
# Confluence

Full $\beta$ reduction has this property:



# Confluence

Full $\beta$ reduction has this property:




## 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 $\beta$ 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:

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

# “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 $\alpha$-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.

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

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.

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

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.

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

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.

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

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:*

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

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

# Encodings

The pure $\lambda$-calculus contains only functions as values. It is not exactly easy to write large or interesting programs in the pure $\lambda$-calculus.

We can however *encode* objects, such as booleans, and integers.

# Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$\begin{aligned} \text{AND TRUE FALSE} &= \text{FALSE} \\ \text{NOT FALSE} &= \text{TRUE} \\ \text{IF TRUE } e_1\ e_2 &= e_1 \\ \text{IF FALSE } e_1\ e_2 &= e_2 \end{aligned}$$

# Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$\begin{aligned} \text{AND TRUE FALSE} &= \text{FALSE} \\ \text{NOT FALSE} &= \text{TRUE} \\ \text{IF TRUE } e_1\ e_2 &= e_1 \\ \text{IF FALSE } e_1\ e_2 &= e_2 \end{aligned}$$

Let's start by defining TRUE and FALSE:

$$\begin{aligned} \text{TRUE} &\triangleq \\ \text{FALSE} &\triangleq \end{aligned}$$

# Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$\begin{aligned} \text{AND TRUE FALSE} &= \text{FALSE} \\ \text{NOT FALSE} &= \text{TRUE} \\ \text{IF TRUE } e_1\ e_2 &= e_1 \\ \text{IF FALSE } e_1\ e_2 &= e_2 \end{aligned}$$

Let's start by defining TRUE and FALSE:

$$\begin{aligned} \text{TRUE} &\triangleq \lambda x.\, \lambda y.\, x \\ \text{FALSE} &\triangleq \lambda x.\, \lambda y.\, y \end{aligned}$$

# Booleans

We want the function IF to behave like

$\lambda b.\ \lambda t.\ \lambda f.$ if $b$ is our term TRUE then $t$, otherwise $f$

# Booleans

We want the function IF to behave like

$$\lambda b.\ \lambda t.\ \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$$

# Booleans

We want the function IF to behave like

$$\lambda b.\ \lambda t.\ \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$$

We can also write the standard Boolean operators.

$$\text{NOT} \triangleq$$

$$\text{AND} \triangleq$$

$$\text{OR} \triangleq$$

# Booleans

We want the function IF to behave like

$$\lambda b.\ \lambda t.\ \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$$

We can also write the standard Boolean operators.

$$\text{NOT} \triangleq \lambda b.\ b\ \text{FALSE}\ \text{TRUE}$$
$$\text{AND} \triangleq \lambda b_1.\ \lambda b_2.\ b_1\ b_2\ \text{FALSE}$$
$$\text{OR} \triangleq \lambda b_1.\ \lambda b_2.\ b_1\ \text{TRUE}\ b_2$$

# Church Numerals

Let's encode the natural numbers!

We'll write $\overline{n}$ for the encoding of the number $n$. The central function we'll need is a *successor* operation:

$$\text{SUCC}\ \overline{n} = \overline{n+1}$$

# Church Numerals

Church numerals encode a number $n$ as a function that takes $f$ and $x$, and applies $f$ to $x$ $n$ times.

$$\begin{aligned}\overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\ \overline{1} &\triangleq \lambda f.\, \lambda x.\, f\, x \\ \overline{2} &\triangleq \lambda f.\, \lambda x.\, f\,(f\, x)\end{aligned}$$

# Church Numerals

Church numerals encode a number $n$ as a function that takes $f$ and $x$, and applies $f$ to $x$ $n$ times.

$$\begin{aligned}\overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\ \overline{1} &\triangleq \lambda f.\, \lambda x.\, f\, x \\ \overline{2} &\triangleq \lambda f.\, \lambda x.\, f\,(f\, x)\end{aligned}$$

We can write a successor function that “inserts” another application of $f$:

$$\text{SUCC} \triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x)$$

# Addition

Given the definition of SUCC, we can define addition. Intuitively, the natural number $n_1 + n_2$ is the result of applying the successor function $n_1$ times to $n_2$.

$$\text{PLUS} \triangleq$$

# Addition

Given the definition of SUCC, we can define addition. Intuitively, the natural number $n_1 + n_2$ is the result of applying the successor function $n_1$ times to $n_2$.

$$\text{PLUS} \triangleq \lambda n_1.\, \lambda n_2.\, n_1 \text{ SUCC } n_2$$

# Church Numerals

We can define more functions on integers:

$$\begin{aligned} \text{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x) \\ \text{PLUS} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{SUCC}\ n_2 \end{aligned}$$

# Church Numerals

We can define more functions on integers:

$$
\begin{aligned}
\text{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x) \\
\text{PLUS} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{SUCC}\ n_2 \\
\text{TIMES} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ (\text{PLUS}\ \text{n}_2)\ \overline{0}
\end{aligned}
$$

# Church Numerals

We can define more functions on integers:

$$
\begin{aligned}
\text{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x) \\
\text{PLUS} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{SUCC}\ n_2 \\
\text{TIMES} &\triangleq \lambda n_1.\, \lambda n_2.\, n_1\ (\text{PLUS}\ \text{n}_2)\ \overline{0} \\
\text{ISZERO} &\triangleq \lambda n.\, n\ (\lambda z.\, \text{FALSE})\ \text{TRUE}
\end{aligned}
$$