CS242

2015

Fundamentals

Edward Z. Yang

Aside: Fixpoint on streams

Function f

$1, 2, 3, 4, \ldots$

$\emptyset, 1, 2, 3, 4, \ldots$

Fixpoint is repeat $\emptyset$

$\ldots$

$\emptyset, \ldots$

$\emptyset, \emptyset, \ldots$

$\emptyset, \emptyset, \emptyset \ldots$

Aside: Fixpoint on streams

Function g

$1, 2, 3, 4, \ldots$

$\emptyset, 1, 3, 5, 7, \ldots$

Fixpoint is Fibonaccis

$\ldots$

$\emptyset, 1, \ldots$

$\emptyset, 1, 1, \ldots$

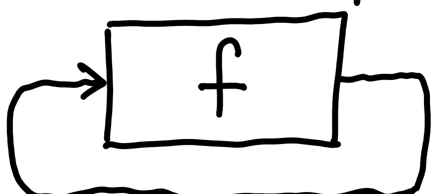
$\emptyset, 1, 1, 2, \ldots$

$\emptyset, 1, 1, 2, 3, \ldots$

Aside: Fixpoint on streams

One strategy: Start w/ empty stream ε,
then $\underbrace{f(f(\cdots f(f}_{\infty}(\varepsilon))\cdots))$
is fixpoint (Problem: stack blow up!)

Idea: $f(s)$ is both a producer and a consumer of s. Wire up with self:



Blackboard

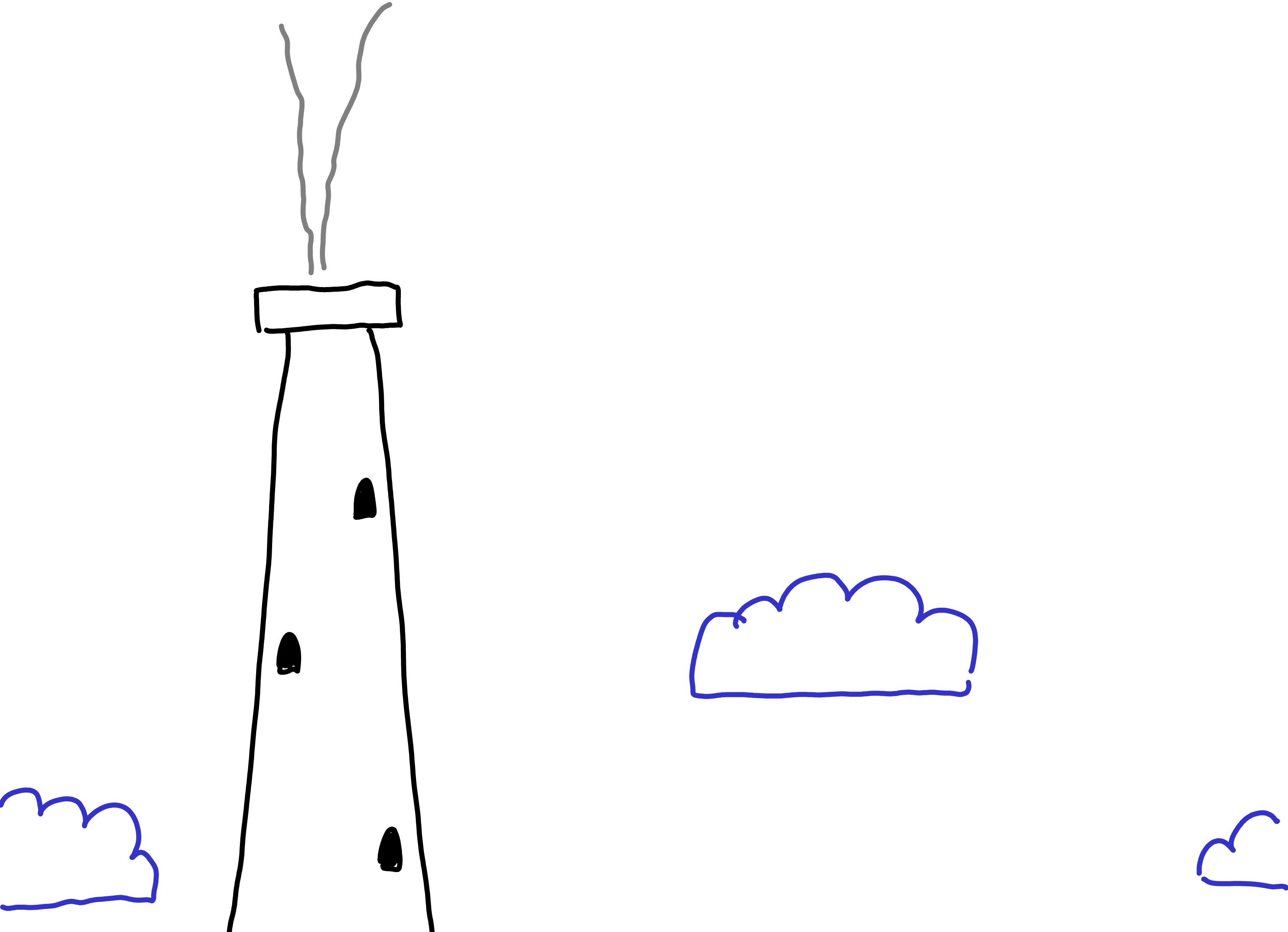
$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2$$

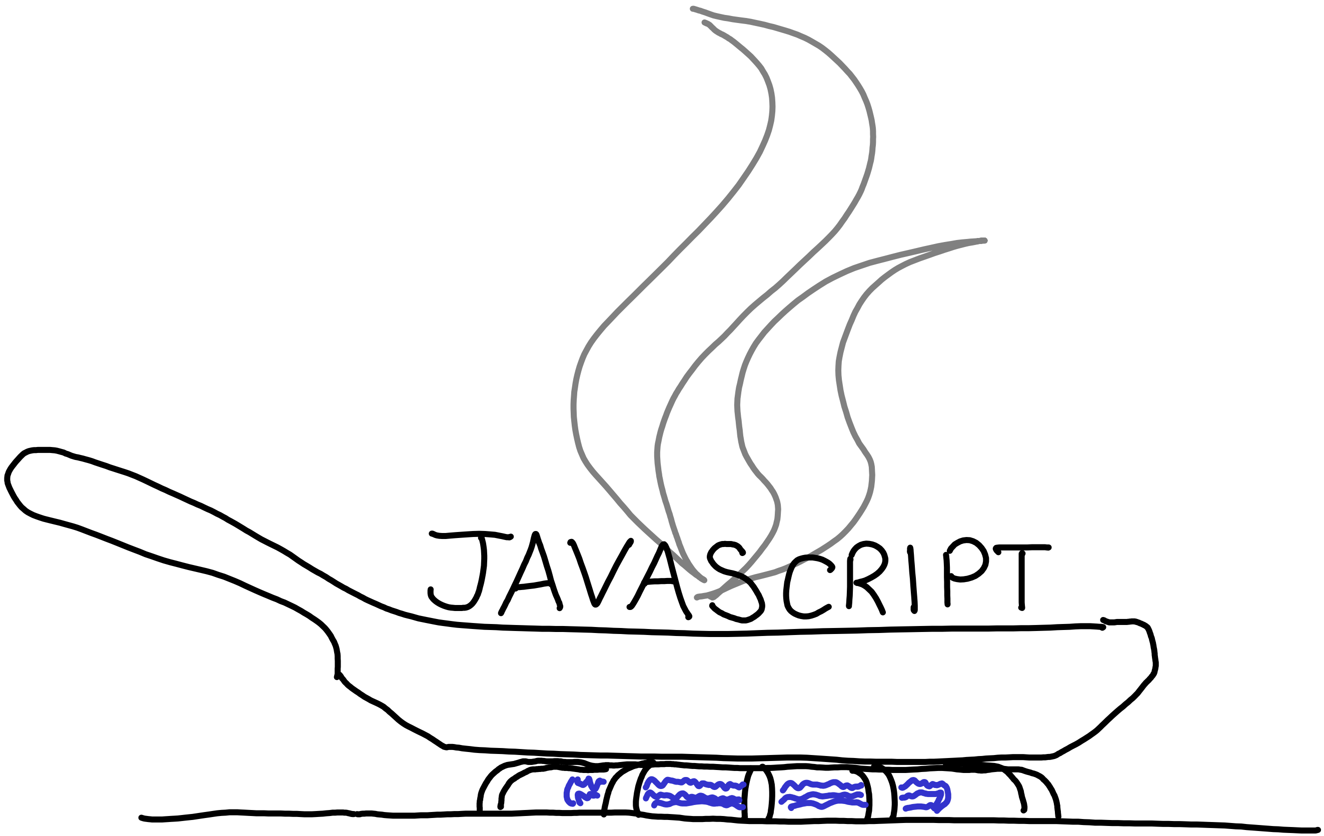
JavaScript

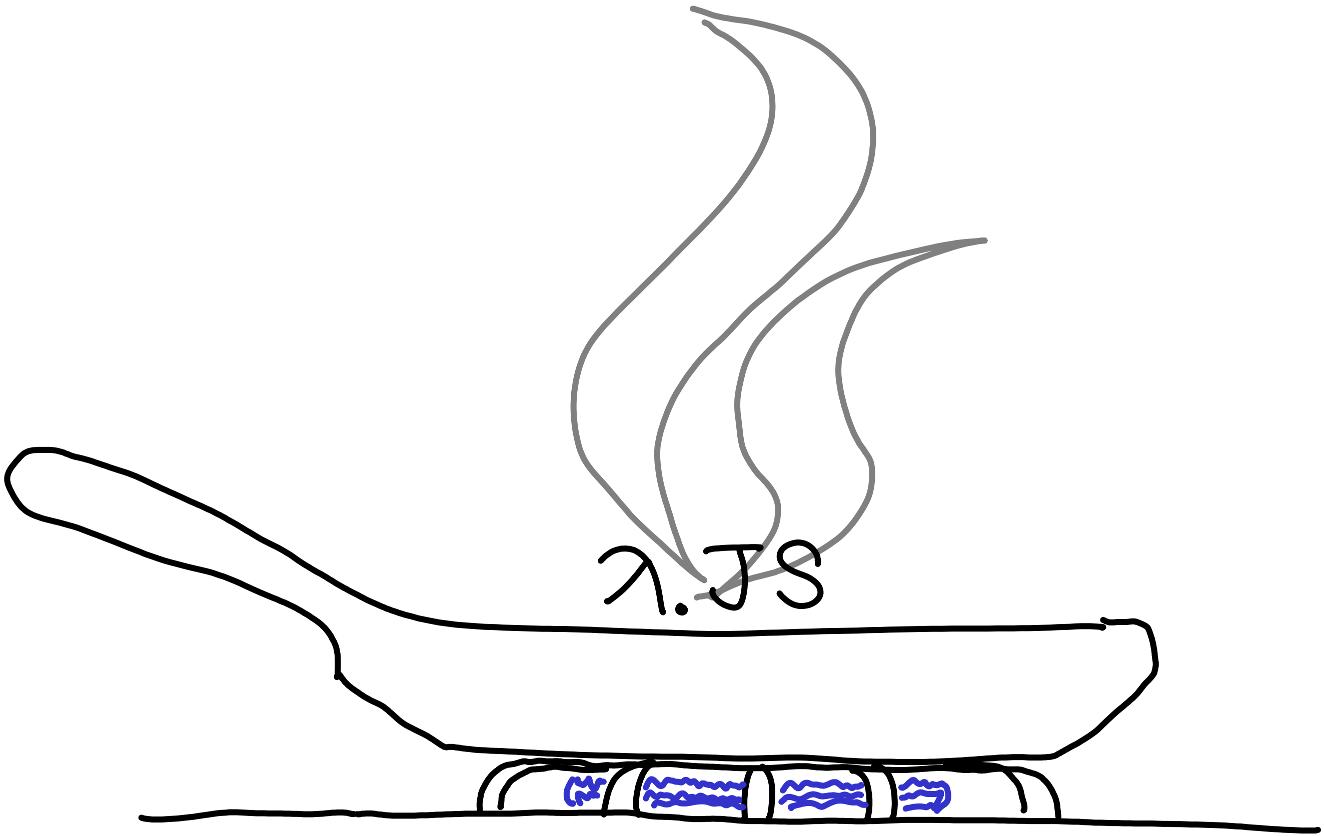
$$
\begin{aligned}
e ::= {} & x \\
\mid {} & \texttt{function}(x)\ \{\ \texttt{return}\ e_1\ \} \\
\mid {} & e_1(e_2)
\end{aligned}
$$

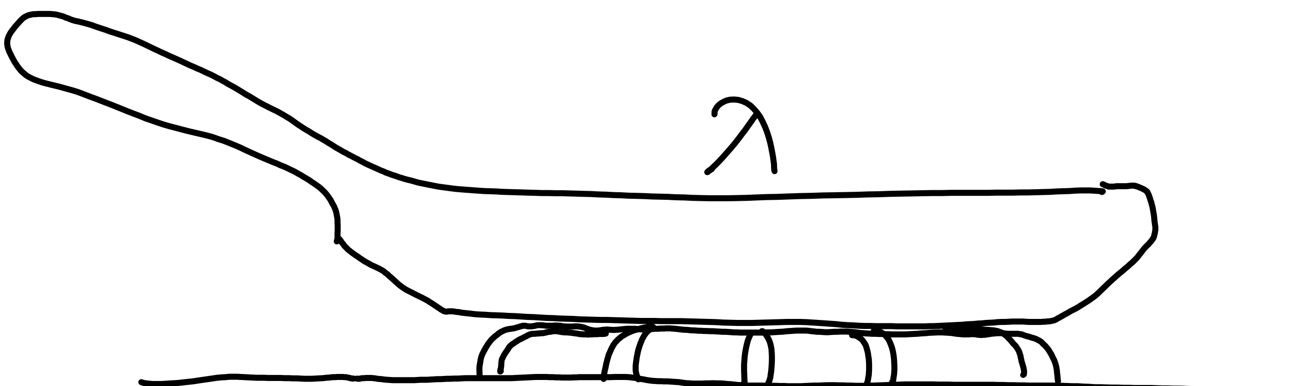
Haskell

$$
\begin{aligned}
e ::= {} & x \\
\mid {} & \backslash x \texttt{ -> } e \\
\mid {} & e_1\ e_2
\end{aligned}
$$




JAVASCRIPT


λ.JS



λ

binders
capture-avoiding substitution (macros, optimizers)
Church encodings (folds, data is code)

λ + evaluation strategy

call-by-value
call-by-name

(not today)

λ + type system

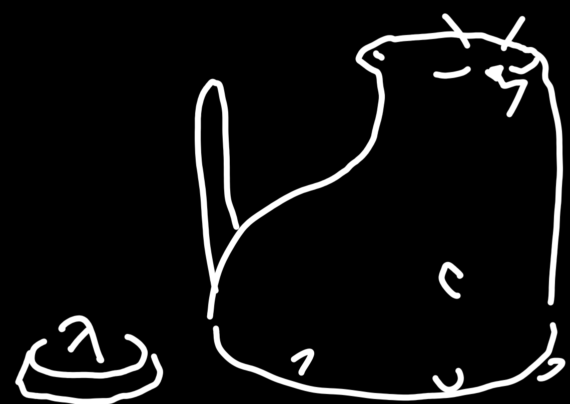
simply-typed lambda calculus
polymorphic lambda calculus
dependent types
every research paper ever

Roadmap

$\boxed{\text{the } \lambda\text{-calculus}}$

capture-avoiding substitution

evaluation order



Recap

$$e ::= x \mid \lambda x.\, e \mid e_1\, e_2$$

Example terms:

$$(\lambda x.\, (2+x))$$ (add 2)

$$(\lambda x.\, (2+x))\ 5 \Rightarrow 7$$

$$(\lambda f.\, (f\ 3))\ (\lambda x.\, (x+1)) \Rightarrow 4$$

higher order function

Recap: Substitution

$$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1)$$

$$\to_\beta \lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ x)$$

$$\to_\beta \lambda x.\ (\lambda y.\ y+1)\ (x+1)$$

$$\to_\beta \lambda x. (x+1)+1$$

Recap: Closures

$$((\lambda x. (\lambda y. x)) \; 2) \; 3$$

$$\to_\beta (\lambda y. 2) \; 3$$

$$\to_\beta 2$$

returned function has x substituted

Using the λ calculus: Syntax

$$\lambda x\, y.\, e \equiv \lambda x. (\lambda y.\, e)$$

Left associative application:

different:

$$f\ x\ y \equiv (f\ x)\ y \neq f\ (x\ y)$$

different:

$$\lambda x.\, f\ x \equiv \lambda x. (f\ x) \neq (\lambda x.\, f)\ x$$

(like Haskell: `\x y -> e` ≡ `\x -> (\y -> e)`)

Using the λ calculus: Declarations

```
function f(x) {
    return x+2;
}
f(f(3));
```

$\Rightarrow$ desugar! $$(\lambda f.\ \overbrace{f\ (f\ 3)}^{\text{block body}})\ \underbrace{(\lambda x.\ x+2)}_{\text{definition of } f}$$

$$\text{let } x = e_1 \text{ in } e_2 \quad \Rightarrow \quad (\lambda x.\ e_2)\ e_1$$

Bound and Free variables

$$(\lambda x . x)$$

Bound Variable
(a closed term)

$$(\lambda x . y)$$

Free Variable
(an open term)

Bound and Free variables

α-conversion

$$(\lambda z . z)$$

name doesn't matter
has no free variables

$$(\lambda z . y)$$

name matters!
y is a free variable

"I am not a number,
I am a free variable!"

Bound and Free variables

function(x) { return x + y }

let $x = e$ in x

Jane hit herself

$\int (x + y)\, dx$

$\forall x.\ P(x)$

$\sum_i x_i$

Bound and Free variables summary

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

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

$$FV(\lambda x.e) = FV(e) \underbrace{\setminus \{x\}}_{\text{remove } x \text{ from set}}$$

α-conversion: rename bound variables (without capturing free variables)

$$(\lambda x.y) \neq_\alpha (\lambda y.y)$$

α-equivalence: equality up to α-conversion

Aside: on the subject of equivalence

We agree $\lambda x . x =_\alpha \lambda y . y$

How about $\lambda x . f\, x \stackrel{?}{=} f$

Aside: on the subject of equivalence

We agree $\lambda x.x =_\alpha \lambda y.y$

How about $\lambda x.f\,x \overset{?}{=}_\eta f$ ✓

Eta-equivalence

Aside: on the subject of equivalence

We agree $\lambda x.x =_\alpha \lambda y.y$

metavariable

How about $\lambda x.\, E\ x \overset{\checkmark}{=}_\eta E$

Eta-equivalence

Any E if $x \notin FV(E)$

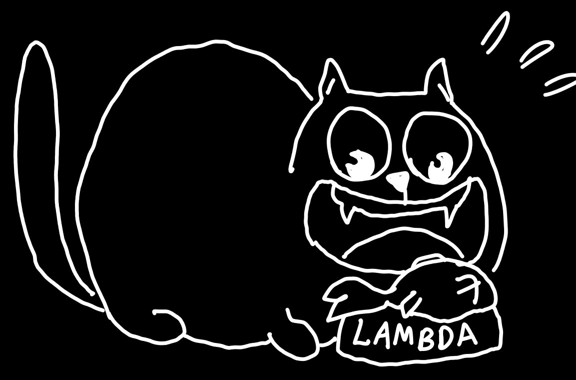
(e.g. if $E \equiv x$, $\lambda x.\, x\ x \neq_\eta x$)

Roadmap

the λ-calculus: binders

$\boxed{\text{capture-avoiding substitution}}$

evaluation order



Substitution is useful

- Evaluation strategy (conceptual, not so great for implementation)
- Optimization / Macros [SPJ'02]

can't run because we don't know a or b

$$\begin{aligned}&\text{let } x = a+b \text{ in}\\ &\quad \text{let } a = 7 \text{ in}\\ &\qquad x + a\end{aligned}$$

but would like to inline x

How do we compute on λ-terms?

compute!

$$\underbrace{(\lambda x.\, e_1)\; e_2}_{\text{redex}} \longrightarrow_\beta \underbrace{e_1\,[x \mapsto e_2]}_{\text{substitution}}$$

β-reduction

Name capture

$$\boxed{\text{Recall} \quad \text{let } x = e_1 \text{ in } e_2 \equiv (\lambda x. e_2)\ e_1}$$

$$\begin{array}{l} \text{let } x = a+b \text{ in} \\ \quad \text{let } a = 7 \text{ in} \\ \qquad x + a \end{array} \quad \not\Rrightarrow \quad \begin{array}{l} \text{let } a = 7 \text{ in} \\ \quad (a+b) + a \end{array}$$

obviously wrong

Name capture

Recall $\text{let } x = e_1 \text{ in } e_2$

$\equiv (\lambda x. e_2)\ e_1$

let x = a+b in
  let a = 7 in
    x + a

✓ ⟹

let s796 = 7 in
  (a+b) + s769

Some "fresh" new variable

Capture-avoiding substitution

Idea: Rename bound variables (α-convert them) so that they don't capture free variables

Capture-avoiding substitution

$$x[x \mapsto e] = e$$

$$y[x \mapsto e] = y$$

$$(e_1\ e_2)[x \mapsto e] = e_1[x \mapsto e]\ \ e_2[x \mapsto e]$$

$$(\lambda x.e_1)[x \mapsto e] = \lambda x.e_1$$

$$(\lambda x.e_1)[y \mapsto e] = \lambda x.e_1[y \mapsto e] \text{ if } x \notin FV(e)$$

$$(\lambda y.e_1)[x \mapsto e] = \lambda y'.\, e_1[y \mapsto y'][x \mapsto e]$$

where y' is fresh

Capture-avoiding substitution

$$x[x \mapsto e] = e$$

$$y[x \mapsto e] = y$$

$$(e_1\ e_2)[x \mapsto e] = e_1[x \mapsto e]\ \ e_2[x \mapsto e]$$

$$(\lambda x.e_1)[x \mapsto e] = \lambda x.e_1$$

$$(\lambda x.e_1)[y \mapsto e] = \lambda x.e_1[y \mapsto e] \text{ if } x \notin FV(e)$$

$$(\lambda y.e_1)[x \mapsto e] = \lambda y'.\ e_1[y \mapsto y'][x \mapsto e]$$

where $y' \neq x$, $y' \notin FV(e_1)$, and $y' \in FV(e)$

Summary: Equational theory

$\boxed{\alpha}$ $\qquad \lambda x.\, e \longrightarrow_{\alpha} \lambda y.\, e[x \mapsto y]$

where $y \notin FV(e)$

$\boxed{\beta}$ $\qquad (\lambda x.\, e_1)\; e_2 \longrightarrow_{\beta} e_1[x \mapsto e_2]$

$\boxed{\eta}$ $\qquad \lambda x.\, e\, x \longrightarrow_{\eta} e$

where $x \notin FV(e)$

Roadmap

the λ-calculus: binders

capture-avoiding substitution

$\boxed{\text{evaluation order}}$

$$(\lambda x.\, x)\ ((\lambda y.\, y)\ z)$$

$$\begin{tikzcd}
 & (\lambda x.x)\,((\lambda y.y)\ z) \arrow[dl, "\beta"', "\text{outer}"] \arrow[dr, "\text{inner}"', "\beta"] & \\
(\lambda y.y)\ z \arrow[dr, "\beta"'] & & (\lambda x.x)\ z \arrow[dl, "\beta"] \\
 & z &
\end{tikzcd}$$

Does it matter?

Does it matter?

Church-Rosser Theorem:

"If you reduce to a normal form, it doesn't matter what order you do the reductions."

Does it matter?

Church-Rosser Theorem:

"<u>If you reduce to a normal form,</u> it doesn't matter what order you do the reductions."

A curious lambda term called Ω

$$(\lambda x.\, x\, x)\ (\lambda x.\, x\, x)$$

A curious lambda term called Ω

$$(x\,x)[x \mapsto (\lambda x.\, x\,x)]$$

A curious lambda term called Ω

$$(\lambda x.\, x\; x)\; (\lambda x.\, x\; x)$$

Deja vu!

Ω has no normal form

$$\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \Omega \to \Omega \to$$

$$(\lambda x . y)\,\Omega$$

$$(\lambda x.\, y)\, \Omega \nearrow y$$

$$
\begin{tikzcd}
 & y \\
(\lambda x . y)\ \Omega \arrow[ur] \arrow[dr] & \\
 & (\lambda x . y)\ \Omega
\end{tikzcd}
$$

$$
\begin{tikzcd}
& & y & & \\
(\lambda x.y)\,\Omega \arrow[urr] \arrow[drr] & & & y & \\
& & (\lambda x.y)\,\Omega \arrow[ur] \arrow[rr] & & (\lambda x
\end{tikzcd}
$$

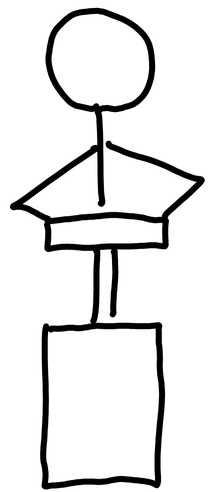
ok, evaluation order might be important

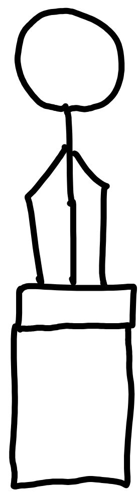
Call-by-value (ala JavaScript)

$$
\begin{aligned}
& e_1 \; e_2 \\
\longrightarrow_\beta^* \; & (\lambda x. e_1') \; e_2 \\
\longrightarrow_\beta^* \; & (\lambda x. e_1') \; n \\
\longrightarrow_\beta \; & e_1'[x \mapsto n] \longrightarrow_\beta^* \cdots
\end{aligned}
$$

Call-by-value

$$(\lambda x.\, y)\, \Omega \longrightarrow_{\beta} (\lambda x.\, y)\, \Omega \longrightarrow$$






BOM!

Call-by-name (ala Haskell***)

$$e_1 \; e_2$$

$$\longrightarrow_\beta^* (\lambda x. e_1') \; e_2$$

— (skip) —

$$\longrightarrow_\beta e_1'[x \mapsto e_2] \longrightarrow_\beta^* \cdots$$

Call-by-name

$$(\lambda x.y)\,\Omega \longrightarrow_\beta y$$

only do what is absolutely necessary!

Summary

λ-term may have many redexes

evaluation order says which redex to evaluate

evaluation not guaranteed to find normal form

CBV : evaluate function & arguments
before β-reducing

CBN : evaluate function, then β-reduce

Roadmap

the λ-calculus: binders

capture-avoiding substitution

evaluation order

Conclusion

λ-calculus = Formal System

Conclusion

$$e ::= \lambda x.e \mid e\,e \mid x$$

binders show up everywhere!

$$Y = \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

$$\text{true} = \lambda x.\lambda y.x$$

$$\text{false} = \lambda x.\lambda y.y$$

$$\text{cond} = \lambda b.\lambda t.\lambda f.\ b\,t\,f$$