CSCI-UA 490

Lambda Calculus

Edward Z. Yang

Blackboard

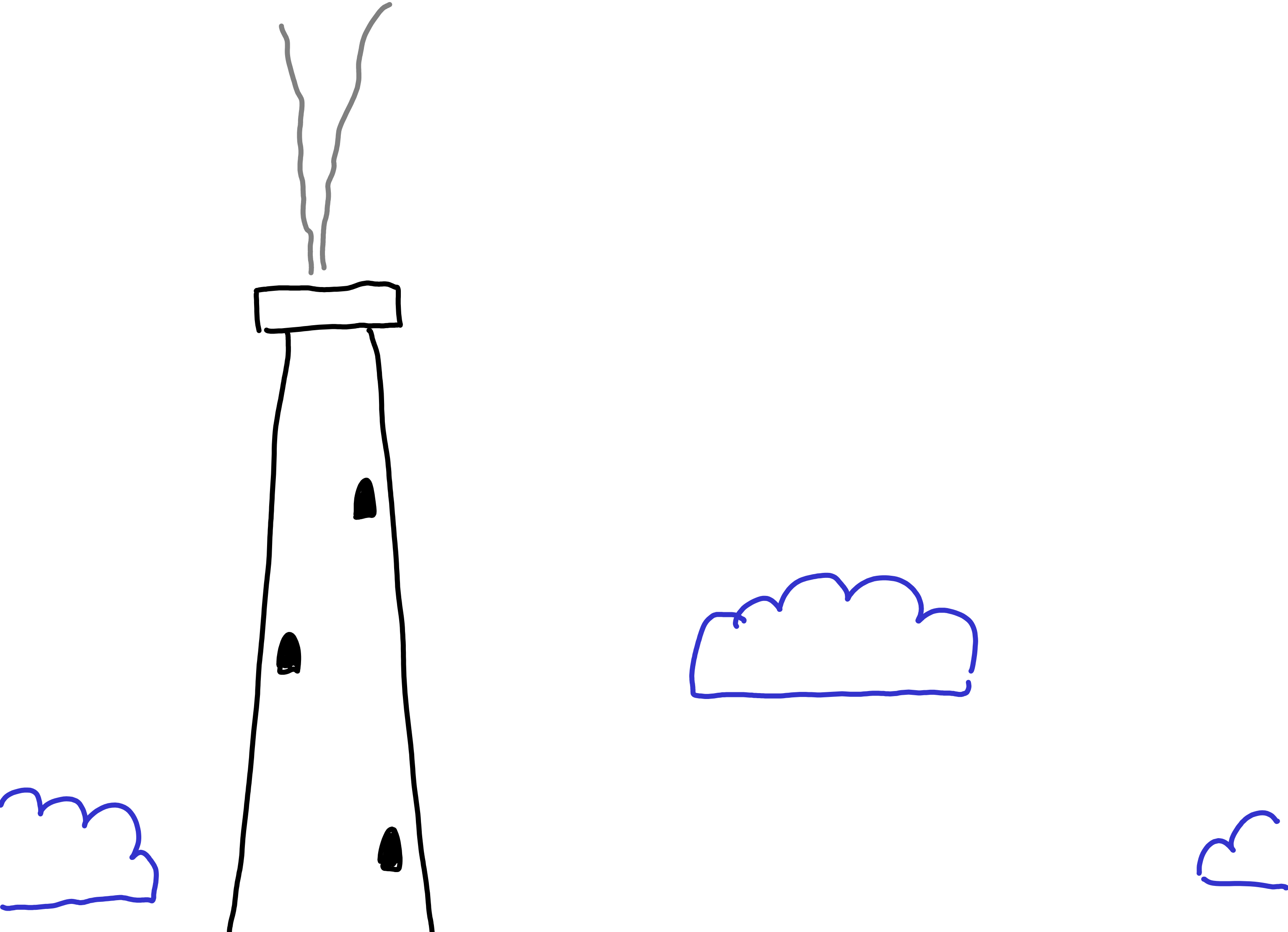
$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2$$

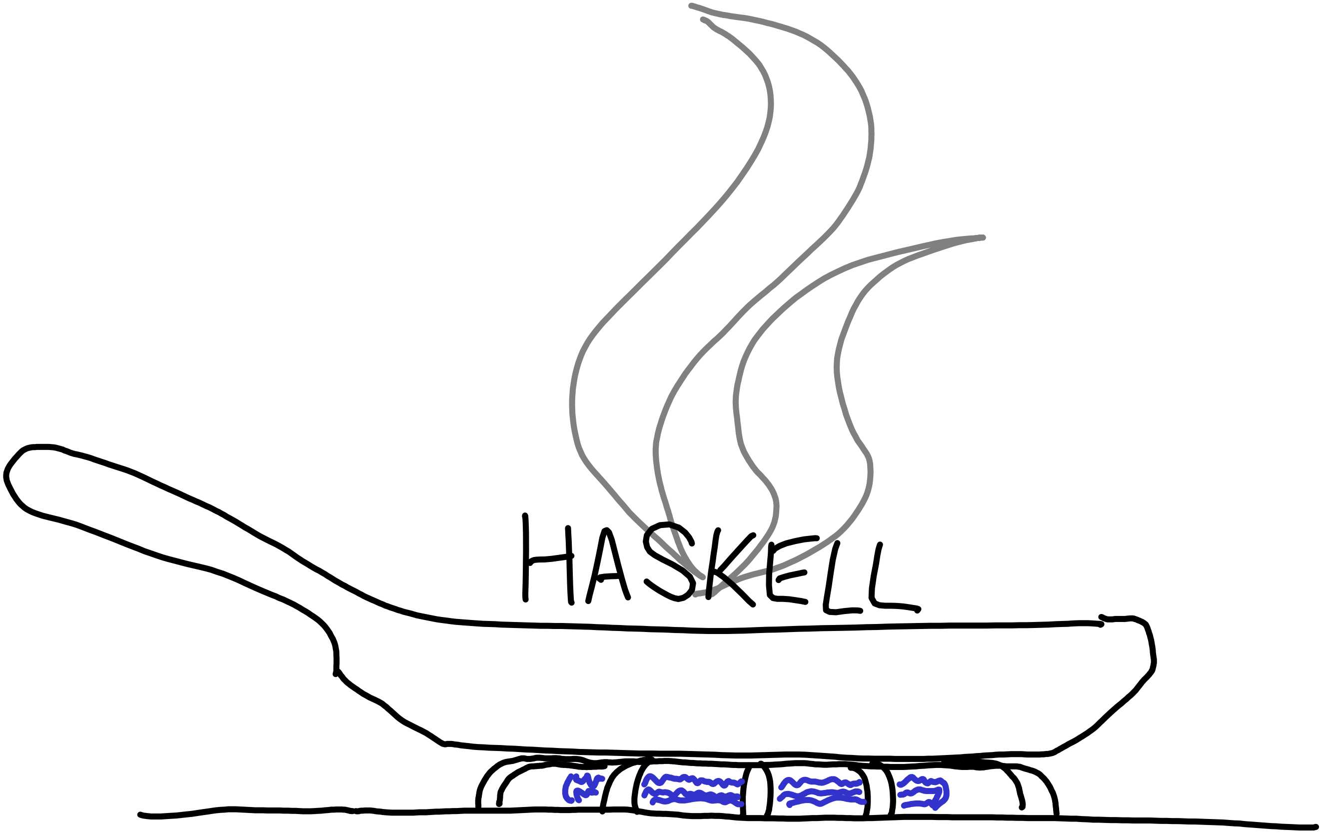
JavaScript

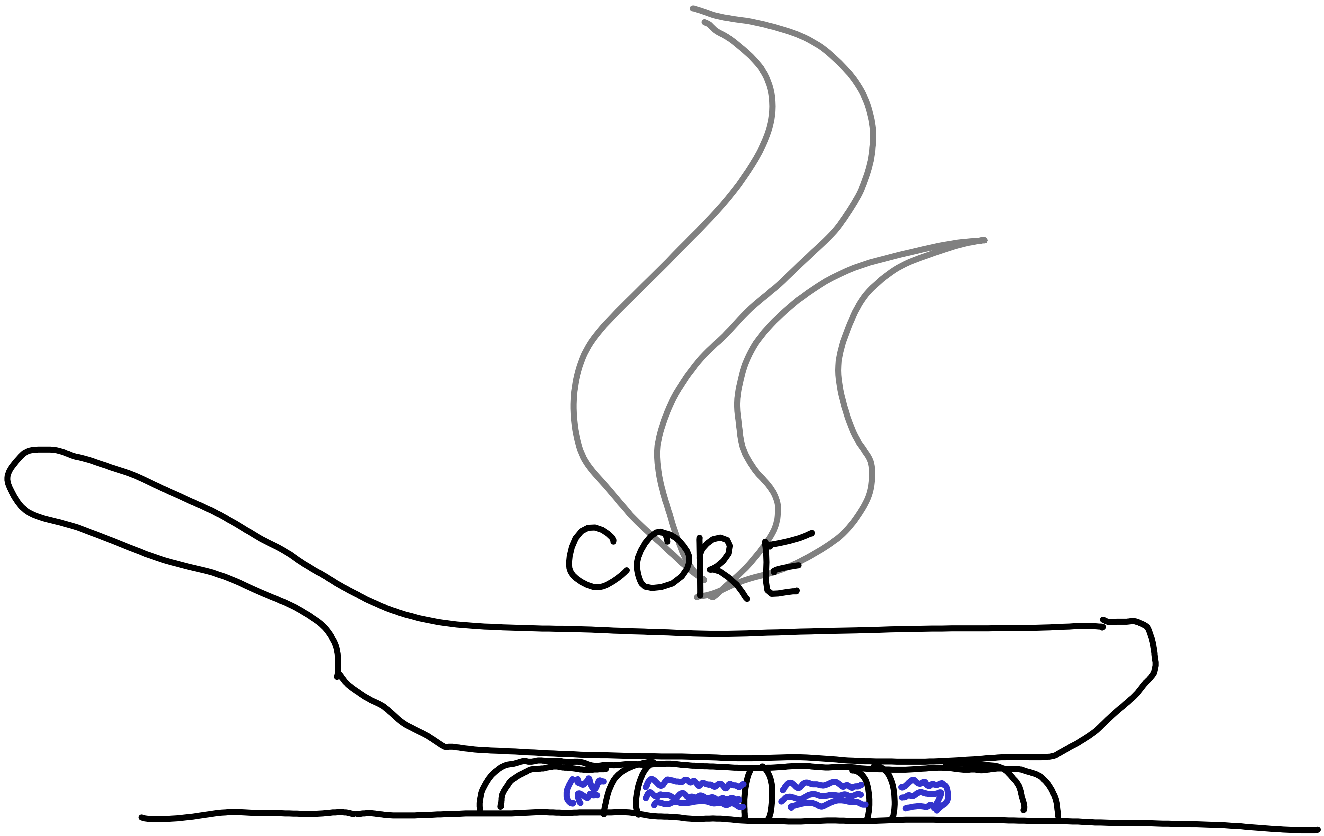
$$\begin{aligned} e ::=\ & x \\ \mid\ & \texttt{function}(x)\ \{\ \texttt{return}\ e_1\ \} \\ \mid\ & e_1(e_2) \end{aligned}$$

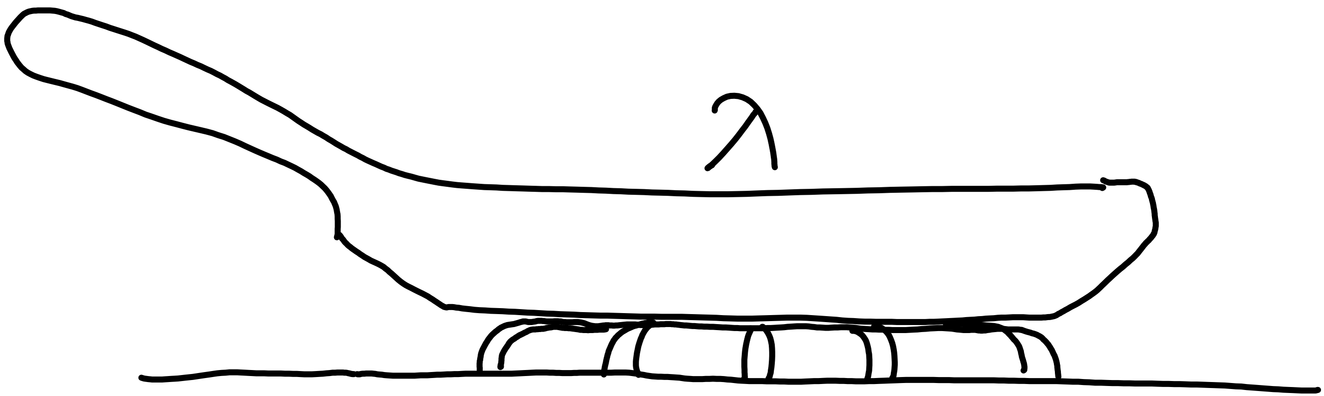
Haskell

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




HASKELL


CORE


λ

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 PL 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` $\equiv$ `\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

$$\text{function}(x)\ \{\ \text{return}\ x + y\ \}$$

$$\text{let}\ x = e\ \text{in}\ x$$

Jane hit herself

$$\int (x + y)\, dx \qquad \forall x.\ P(x) \qquad \sum_i x_i$$

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

de Bruijn indexes

name doesn't matter...

$$(\lambda z . z)$$

de Bruijn indexes

so get rid of it!

$$(\lambda.\ 0)$$

number of lambdas
to count outwards

de Bruijn indexes

$$\lambda x.\ \lambda y.\ x \implies \lambda.\ \lambda.\ 1$$

$$(\lambda x.\ (\lambda y.\ x)\ x) \implies \lambda.\ (\lambda.\ 1)\ 0$$

only counts enclosing lambdas

de Bruijn indexes

$\lambda x.\lambda y.x$

$\lambda y.\lambda x.y$

$\longrightarrow \lambda.\lambda.1$

structural equivalence

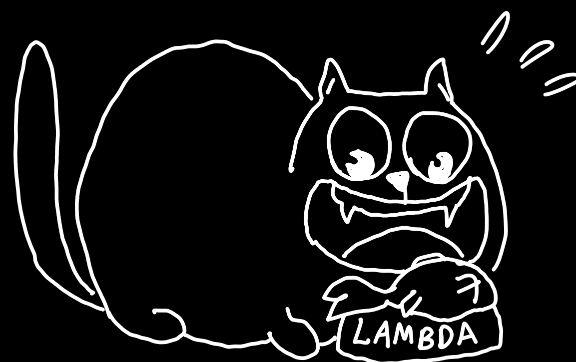
$= \alpha$-equivalence

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

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

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

$$\boxed{\begin{aligned}&\text{Recall}\quad \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\\ &\qquad\equiv (\lambda x.e_2)\ e_1\end{aligned}}$$

$$\begin{aligned}&\mathtt{let}\ x = a+b\ \mathtt{in}\\ &\quad \mathtt{let}\ a = 7\ \mathtt{in}\\ &\qquad x + a\end{aligned} \quad\not\Rightarrow\quad \begin{aligned}&\mathtt{let}\ a = 7\ \mathtt{in}\\ &\quad (a+b) + a\end{aligned}$$

obviously wrong

Name capture

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

$\text{let } x = a + b \text{ in}$
$\quad \text{let } a = 7 \text{ in}$
$\qquad x + a$

$\Longrightarrow$

$\text{let } s796 = 7 \text{ in}$
$\quad (a + b) + s769$

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

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

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

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

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

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

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

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 $y' \notin \{x\} \cup FV(e_1) \cup FV(e)$

Summary: Equational theory

$\boxed{\alpha}$ $\quad \lambda x.e \equiv_\alpha \lambda y.e[x \mapsto y]$

where $y \notin FV(e)$

$\boxed{\beta}$ $\quad (\lambda x.e_1)\, e_2 \equiv_\beta e_1[x \mapsto e_2]$

$\boxed{\eta}$ $\quad \lambda x.e\, x \equiv_\eta e$

where $x \notin FV(e)$

Roadmap

the λ-calculus: binders

capture-avoiding substitution

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

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

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

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

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

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

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

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

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

ok, evaluation order might be important

Call-by-value (ala JavaScript)

$$e_1 \; e_2$$

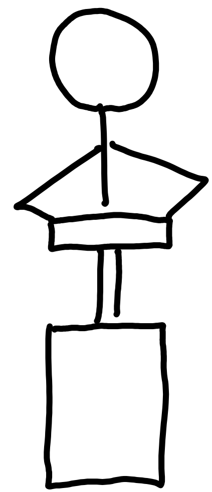
$$\xrightarrow{}{}^{*}_{\beta} \; (\lambda x. e_1') \; e_2$$

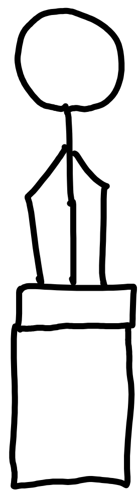
$$\xrightarrow{}{}^{*}_{\beta} \; (\lambda x. e_1') \; n$$

$$\xrightarrow{}_{\beta} \; e_1'[x \mapsto n] \xrightarrow{}{}^{*}_{\beta} \cdots$$

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) —

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

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!

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

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

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

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

Extra topics

- Locally nameless style
- Other evaluation strategies
- Operational semantics