Lambda calculus (cont)

Deian Stefan

(adopted from my & Edward Yang's CSE242 slides)

Logistics

- Assignments:
  - HW 1 is out and due this week (Sunday)
  - There will be one more homework on functions
  - After this: 1 homework / general topic area
- Podcasting: no video while projector is broken
  - Sorry :(
- Come to section and office hours!

Questions

- How are you finding PA1?
  - A: easy, B: okay, C: hard, D: wtf is PA1?

Questions

- How are you finding HW1?
  - A: easy, B: okay, C: hard

Questions

- How are you finding the pace of the lectures?
  - A: too slow, B: it works for me, C:too fast

Today

- Recall syntax of λ calculus
- Semantics of λ calculus
  - Recall free and bound variables
  - Substitution
  - Evaluation order

Review

- λ-calculus syntax: $e ::= x \mid \lambda x.e \mid e_1\ e_2$
  - ➤ Is λ(x+y).3 a valid term? (A: yes, B: no)
  - ➤ Is λx. (x x) a valid term? (A: yes, B: no)
  - ➤ Is λx. (x) y a valid term? (A: yes, B: no)

More compact syntax (HW)

- Function application is left associative
  - $e_1\ e_2\ e_3 \stackrel{\text{def}}{=} (e_1\ e_2)\ e_3$
- Lambdas binds all the way to right: only stop when you find unmatched closing paren ')'
  - $\lambda x.\lambda y.\lambda z.e \stackrel{\text{def}}{=} \lambda x.(\lambda y.(\lambda z.e))$

More on syntax

- Write the parens: λx.x x
  - **A: λx.(x x)**
  - B: (λx.x) x

More on syntax

- Write the parens: λy.λx.x x =
  - A: λy.(λx.x) x
  - **B: λy.(λx.(x x))**
  - C: (λy.(λx.x)) x

More on syntax

- Is $(\lambda y.\lambda x.x)\ x = \lambda y.\lambda x.x\ x$?
  - A: yes
  - **B: no**

How do we compute in λ calculus?

How do we compute in λ calculus?

- Substitution!
  - When do we use substitution?
  - What's the challenge with substitution?

Example terms

- Reduce (λx.(2 + x)) 5
- Reduce (λx.(λy.2) 3) 5 → (λx. 2) 5 → 2
- Reduce (board): ((λx.(λy.2)) 3) 5 → ((λy.2) 5) → 2
- Reduce: (λx.λy.λz.y+3) 4 5 6

Even more compact syntax

- Can always variables left of the .
  - $\lambda x.\lambda y.\lambda z.e \stackrel{\text{def}}{=} \lambda xyz.e$
- This makes the term look like a 3 argument function
  - Can implement multiple-argument function using single-argument functions: called currying (bonus)
- We won’t use this syntax, but you may see in the wild

Why is substitution hard?

- What does this reduce to if we do it blindly?
  - let x = a+b in
      let a = 7 in
        x + a

- Recall: let $x = e_1$ in $e_2 \stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1$
  - Reduce $(\lambda x.\ (\lambda a.\ x + a)\ 7)\ (a+b)$

How do we fix this?

- Renaming!
  - A: rename all free variables
  - B: rename all bound variables

Def: free variables (recall)

- If a variable is not bound by a λ, we say that it is **free**
  - e.g., y is free in $\lambda x.(x+y)$
  - e.g., x is bound in $\lambda x.(x+y)$
- We can compute the free variables of any term:
  - $FV(x) = \{x\}$
  - $FV(\lambda x.e) = FV(e) \setminus \{x\}$
  - $FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

think: build out!

Def: free variables (recall)

- If a variable is not bound by a λ, we say that it is **free**
  - e.g., y is free in $\lambda x.(x+y)$
  - e.g., x is bound in $\lambda x.(x+y)$
- We can compute the free variables of any term:
  - $FV(x) = \{x\}$
  - $FV(\lambda x.e) = FV(e) \setminus \{x\}$
  - $FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

think: build out!

Def: free variables (recall)

- If a variable is not bound by a λ, we say that it is **free**
  - e.g., y is free in $\lambda x.(x+y)$
  - e.g., x is bound in $\lambda x.(x+y)$
- We can compute the free variables of any term:
  - $FV(x) = \{x\}$
  - $FV(\lambda x.e) = FV(e) \setminus \{x\}$
  - $FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

think: build out!

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
  - $x[x{:=}e] = e$
  - $y[x{:=}e] = y$ if $y \neq x$
  - $(e_1\ e_2)[x := e] = (e_1[x := e])\ (e_2[\ x{:=}\ e])$
  - $(\lambda x.e_1)[x := e] = \lambda x.e_1$
  - $(\lambda y.e_1)[x := e_2] = \lambda y.e_1[x := e_2]$ if $y \neq x$ and $y \notin FV(e_2)$

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
  - $x[x:=e] = e$
  - $y[x:=e] = y$ if $y \neq x$
  - $(e_1\ e_2)[x := e] = (e_1[x := e])\ (e_2[x := e])$
  - $(\lambda x.e_1)[x := e] = \lambda x.e_1$
  - $(\lambda y.e_1)[x := e_2] = \lambda y.e_1[x := e_2]$ if $y \neq x$ and $y \notin FV(e_2)$
    - Why the if?

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
  - $x[x:=e] = e$
  - $y[x:=e] = y$ if $y \neq x$
  - $(e_1\ e_2)[x := e] = (e_1[x := e])\ (e_2[x := e])$
  - $(\lambda x.e_1)[x := e] = \lambda x.e_1$
  - $(\lambda y.e_1)[x := e_2] = \lambda y.e_1[x := e_2]$ if $y \neq x$ and $y \notin FV(e_2)$
    - Why the if? If y is free in e_2 this would capture it!

Lambda calculus: equational theory

- α-renaming or α-conversion
  - $\lambda x.e = \lambda y.e[x:=y]$ where $y \notin FV(e)$
- β-reduction
  - $(\lambda x.e_1)\ e_2 = e_1\ [x:=e_2]$
- η-conversion
  - $\lambda x.(e\ x) = e$ where $x \notin FV(e)$
- We define our $\rightarrow$ relation using these equations!

Back to old example

Back to old example

- Instead of 1, let's add x to argument (and do it 2x):
  - $(\lambda f.(\lambda x.\ f\ (f\ x)))\ (\lambda y.y+x)$

Back to old example

- Instead of 1, let’s add x to argument (and do it 2x):
  - (λf.(λx. f (f x))) (λy.y+x)

  $=_\alpha$ (λf.(λz. f (f z))) (λy.y+x)

Back to old example

- Instead of 1, let's add x to argument (and do it 2x):
  - $(\lambda f.(\lambda x.\ f\ (f\ x)))\ (\lambda y.y+x)$

$=_\alpha$ $(\lambda f.(\lambda z.\ f\ (f\ z)))\ (\lambda y.y+x)$

$=_\beta$ $\lambda z.\ (\lambda y.y+x)\ ((\lambda y.y+x)\ z)$

Back to old example

- Instead of 1, let's add x to argument (and do it 2x):
  - $(\lambda f.(\lambda x.\ f\ (f\ x)))\ (\lambda y.y+x)$

$=_\alpha$ $(\lambda f.(\lambda z.\ f\ (f\ z)))\ (\lambda y.y+x)$

$=_\beta$ $\lambda z.\ (\lambda y.y+x)\ ((\lambda y.y+x)\ z)$

$=_\beta$ $\lambda z.\ (\lambda y.y+x)\ (z+x)$

Back to old example

- Instead of 1, let's add x to argument (and do it 2x):
  - (λf.(λx. f (f x))) (λy.y+x)

    $=_\alpha$ (λf.(λz. f (f z))) (λy.y+x)

    $=_\beta$ λz. (λy.y+x) ((λy.y+x) z)

    $=_\beta$ λz. (λy.y+x) (z+x)

    $=_\beta$ λz. z+x+x

Today

- Recall syntax of λ calculus ✓
- Semantics of λ calculus ✓
  - Recall free and bound variables ✓
  - Substitution ✓
  - Evaluation order

Evaluation order

- What should we reduce first in $(\lambda x.x)\ ((\lambda y.y)\ z)$?
  - A: The outer term: $(\lambda y.y)\ z$
  - B: The inner term: $(\lambda x.x)\ z$

Evaluation order

- What should we reduce first in (λx.x) ((λy.y) z)?
  - A: The outer term: (λy.y) z
  - B: The inner term: (λx.x) z
- Does it matter?

Evaluation order

- What should we reduce first in $(\lambda x.x)$ $((\lambda y.y)$ $z)$?
  - A: The outer term: $(\lambda y.y)$ z
  - B: The inner term: $(\lambda x.x)$ z
- Does it matter?
  - No! They both reduce to z!
  - Church-Rosser Theorem: "If you reduce to a normal form, it doesn't matter what order you do the reductions." This is known as confluence.

Does evaluation order really not matter?

Does evaluation order really not matter?

- Consider a curious term called Ω
  - $\Omega \stackrel{\text{def}}{=} (\lambda x.x\ x)\ (\lambda x.x\ x)$

Does evaluation order really not matter?

- Consider a curious term called Ω
  - $\Omega \stackrel{\text{def}}{=} (\lambda x.x\ x)\ (\lambda x.x\ x)$

$=_\beta (x\ x)[\ x := (\lambda x.x\ x)]$

Does evaluation order really not matter?

- Consider a curious term called Ω
  - $\Omega \stackrel{\text{def}}{=} (\lambda x.x\ x)\ (\lambda x.x\ x)$

$=_\beta (x\ x)[\, x := (\lambda x.x\ x)]$

$=_\beta (\lambda x.x\ x)\ (\lambda x.x\ x)$

Does evaluation order really not matter?

- Consider a curious term called Ω
  - $\Omega \stackrel{\text{def}}{=} (\lambda x.x\ x)\ (\lambda x.x\ x)$

$=_\beta\ (x\ x)[\ x := (\lambda x.x\ x)]$

$=_\beta\ (\lambda x.x\ x)\ (\lambda x.x\ x)$

$=\ \Omega$

Deja vu!

$$\Omega \to \Omega \to \Omega \to \Omega \to \Omega \to \Omega \to \Omega$$

(Ω has no normal form)

Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω?

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

Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω?

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

Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω?

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

Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω?

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

Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω?

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

Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω?

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

Does evaluation order really not matter?

- Nope! Evaluation order does matter!

Call-by-value

- Reduce function, then reduce args, then apply
  - e_1 e_2
- JavaScript's evaluation strategy is call-by-value (ish)
  - What does this program do?
    - `(x => 33) ((x => x(x)) (x => x(x)))`

Call-by-value

- Reduce function, then reduce args, then apply
  - $e_1\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ e_2$
- JavaScript's evaluation strategy is call-by-value (ish)
  - What does this program do?
    - `(x => 33) ((x => x(x)) (x => x(x)))`

Call-by-value

- Reduce function, then reduce args, then apply
  - $e_1\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ n$
- JavaScript's evaluation strategy is call-by-value (ish)
  - What does this program do?
    - `(x => 33) ((x => x(x)) (x => x(x)))`

Call-by-value

- Reduce function, then reduce args, then apply
  - $e_1\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ n \rightarrow e_1'[x:=n]$
- JavaScript's evaluation strategy is call-by-value (ish)
  - What does this program do?
    - `(x => 33) ((x => x(x)) (x => x(x)))`

Call-by-value

- Reduce function, then reduce args, then apply
  - $e_1\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ n \rightarrow e_1'[x:=n]$
- JavaScript's evaluation strategy is call-by-value (ish)
  - What does this program do?
    - `(x => 33) ((x => x(x)) (x => x(x)))`
    - `RangeError: Maximum call stack size exceeded`

Call-by-name

- Reduce function, then apply
  - e_1 e_2
- Haskell's evaluation strategy is call-by-name
  - It only does what is absolutely necessary!

Call-by-name

- Reduce function, then apply
  - $e_1\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ e_2$
- Haskell's evaluation strategy is call-by-name
  - It only does what is absolutely necessary!

Call-by-name

- Reduce function, then apply
  - $e_1\ e_2 \to \cdots \to (\lambda x.e_1')\ e_2 \to e_1'[x{:=}e_2]$
- Haskell's evaluation strategy is call-by-name
  - It only does what is absolutely necessary!

Call-by-name

- Reduce function, then apply
  - $e_1\ e_2 \to \cdots \to (\lambda x.e_1')\ e_2 \to e_1'[x:=e_2] \to \cdots$
- Haskell's evaluation strategy is call-by-name
  - It only does what is absolutely necessary!

Summary

- A term may have many redexes (subterms can reduce)
  - Evaluation strategy says which redex to evaluate
  - Evaluation not guaranteed to find normal form
- Call-by-value: evaluate function & args before β reduce
- Call-by-name: evaluate function, then β-reduce

Today

- Recall syntax of λ calculus ✓
- Semantics of λ calculus ✓
  - Recall free and bound variables ✓
  - Substitution ✓
  - Evaluation order ✓

Takeaway

- λ-calculus is a formal system
  - "Simplest reasonable programming language" -Ramsey
  - Binders show up everywhere!
    - Know your capture-avoiding substitution!
    - Macros in HW1
    - JavaScript modules in PA1

Bonus: multi-argument λ's

curry.js