Fundamentals and lambda calculus

Again: JavaScript functions

- JavaScript functions are first-class
  - Syntax is a bit ugly/terse when you want to use functions as values; recall block scoping:

```
(function () {
 // ... do something
}) ();
```

- New version has cleaner syntax called “arrow functions”
  - Semantics not always the same (`this` has different meaning), but for this class should always be safe to use

In this lecture

What is the lambda calculus?

- Simplest reasonable programming language
  - Only has one feature: functions

Why study it?

- Captures the idea of first-class functions
  - Good system for studying the concept of variable binding that appears in almost all languages
- Historically important
  - Competing model of computation introduced by Church as an alternative to Turing machines: substitution (you'll see this today) = symbolic comp
  - Influenced Lisp (thus JS), ML, Haskell, C++, etc.

Why else?

- Base for studying many programming languages
  - You can use lambda calculus and extended it in different ways to study languages and features
    - E.g., we can study the difference between strict languages like JavaScript and lazy ones like Haskell
      - λ + evaluation strategy
    - E.g., we can study different kinds of type systems
      - Simply-typed λ calculus, polymorphic, etc.

Why else?

- Most PL papers describe language models that build on lambda calculus
  - Understanding λ will help you interpret what you are reading in PL research papers
  - Understanding λ will help you get started with other formal/theoretical foundations:
    - Operational semantics
    - Denotational semantics

Before we get started, some terminology

- Syntax (grammar)

- Semantics

- PL implementation: Syntax -> Semantics

Week 1b

- Syntax of λ calculus
- Semantics of λ calculus
  - Informal substitution
  - Free and bound variables
  - Formal substitution
  - Evaluation order

Lambda calculus

- Language syntax (grammar):
  - Expressions: e ::= x | λx.e | e_1 e_2
    - Variables: x
    - Functions or λ abstractions: λx.e
      - This is the same as x => e in JavaScript!
    - Function application: e_1 e_2
      - This is the same as e_1 (e_2) in JavaScript!

Example terms

- λx.(2+x)

LIES! What is this “2” and “+”? (Sugar.)

  - Same as: `x => (2 + x)`

- (λx.(2 + x)) 5

  - Same as: `(x => (2 + x)) (5)`

- (λf.(f 3)) (λx.(x + 1))

  - Same as: `(f => (f (3))) (x => (x+1))`

JavaScript to λ calculus

- Let's look at function composition: `(f ∘ f)(x)`
- In JavaScript:
  - `f => (x => f (f (x)))`
  - `((f => (x => f (f (x)))) (x => x+1)) (4)`
- In λ:

Understanding λ calculus syntax

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

More compact syntax

- 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 ‘)’
  - $\lambda x.\lambda y.\lambda z.e \stackrel{\text{def}}{=} \lambda x.(\lambda y.(\lambda z.e))$
  - Why? Lambda abstraction has lowest precedence!

Understanding compact syntax

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

Understanding compact syntax

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

Understanding compact syntax

- Is (λy.λx.x) x = λy.λx.x x ?
  - A: yes
  - B: no

Parsing rules

- Applications are left associative

- Precedence: application > lambda abstraction

Add parentheses

- λy.λx.x x λz.y λw.w
- λy.λx.(x x) λz.y λw.w
- λy.λx.((x x) λz.y) λw.w
- λy.λx.(((x x) λz.y) λw.w)
- λy.λx.(((x x) λz.y) (λw.w))
  - A: λy.λx.(((x x) (λz.y)) (λw.w))
  - B: λy.λx.(((x x) (λz.y) (λw.w)))

First unmatched closing paren

Even more compact syntax

- Can always move variables left of the period
  - $\lambda x.\lambda y.\lambda z.e \stackrel{\text{def}}{=} \lambda xyz.e$
- This makes the term look like a 3 argument function
  - Can implement multiple-argument function using single-argument functions: called currying (bonus)

Week 1b

- Syntax of λ calculus ✓
- Semantics of λ calculus
  - Informal substitution
  - Free and bound variables
  - Formal substitution
  - Evaluation order

Semantics of λ calculus

- Reduce a term to another as much as we can
  - If we can't reduce it any further, the term is said to be in normal form
- How? Rewrite terms!
  - What does that mean?! Substitution!

Example terms

- Example: (λx.(2 + x)) 5 →
  - In JavaScript: `(x => (2 + x)) (5)` →
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: `(f => (f (3))) (x => (x+1))`

Easy! Pattern for function application: substitute the term you are applying the function to for the argument variable

Substitution (not right)

- Def: Substitution: e_1 [x := e_2]
  - Replace every occurrence of x in e_1 with e_2
- General reduction rule for λ calculus:
  - $(\lambda x.e_1)\ e_2 \rightarrow$
  - Function application rewritten to
    e_1 (the function body) with every x in e_1
    substituted with e_2 (argument)

Simple examples

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

Simple examples (cont)

- Reduce (λx.(λy.2) 3) 5 →

- Reduce: ((λx.(λy.2)) 3) 5 →

- Is (λx.(λy.2) 3) 5 = ((λx.(λy.2)) 3) 5 ?

  - A: yes, B: no

A more complicated example

- Reduce the following:
  - (λx. (λa. x + a) 7) 4
  - (λa. 4 + a) 7
  - 4 + 7
  - 11

Let’s make this even more fun!

- Instead of 4, let’s apply function to (a + 5)
  - (λx. (λa. x + a) 7) (a + 5)
  - (λa. (a + 5) + a) 7
  - (7 + 5) + 7
  - 19

Is this right?
A: yes, B: no

Substitution is surprisingly complex

- Recall our reduction rule for application:
  - $(\lambda x.e_1)\ e_2 \rightarrow e_1\ [x := e_2]$
  - This function application reduces to e_1 (the function body) where every x in e_1 is substituted with e_2 (value we're applying func to)
- Where did we go wrong? When we substituted:
  - $(\lambda x.\ (\lambda a.\ x + a)\ 7)\ (a + 5)$
  - $(\lambda a.\ (a + 5) + a)\ 7$ the a is captured!

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λx.x+1)) 4
  - (λx. (λx.x+1) ((λx.x+1) x)) 4
  - (λx.x+1) ((λx.x+1) 4)
  - (λx.x+1) (4+1)
  - 4+1+1
  - 6

Example to do at home

- Reduce the following
  - (λf.(λx. f (f x))) (λy.y+x)
  - λx. (λy.y+x) ((λy.y+x) x)
  - λx. (λy.y+x) (x+x)
  - λx. (x+x+x)

????

that's not a function that adds x to argument two times

Another way to see the problem

- Syntactic sugar: let x = e_1 in e_2 $\stackrel{\text{def}}{=}$ (λx.e_2) e_1
- Let syntax makes this easy to see:
  - let x = a + 5 in
let a = 7 in
x + a → let x = a + 5 in
let a = 7 in
(a + 5) + a

Fixing the problem

- How can we fix this?

1. Rename variables!

   - let x = a+5 in
     let a = 7 in →
     x + a

2. Do the "dumb" substitution!

Why is this the way to go?

- Def: variable x is bound in λx.(x+y)
  - Can we always rename bound variables? A: yes, B:no
- Yes! Bound variables are just “placeholders”
  - Above: x is not special, we could have used z
  - We say they are equivalent: $\lambda x.(x+y) =_{\alpha} \lambda z.(z+y)$
- Renaming amounts to converting bound variable names to avoid capture: e.g., λx.(x+y) to λz.(z+y)

Can we rename everything?

- Can we rename y in λx.(x+y)? (A: yes, B: no)
  - ➤
- Intuition:
  - ➤
  - ➤

Week 1b

- Syntax of λ calculus ✓
- Semantics of λ calculus
  - Informal substitution ✓
  - Free and bound variables
  - Formal substitution
  - Evaluation order

Let’s think about this more formally

Def: free variables

- If a variable is not bound by a λ, we say that it is **free**
  - e.g., y is free in λx.(x+y)
  - is x free? A: yes, B: no

- We can compute the free variables of any term:
  - FV(x) = {x}
  - FV(λx.e) = FV(e) \ {x}
  - FV(e_1 e_2) =

think: build out!

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
  - $x[x:=e] =$
  - $y[x:=e] =$
  - $(e_1\ e_2)[x := e] =$
  - $(\lambda x.e_1)[x := e] =$
  - $(\lambda y.e_1)[x := e_2] =$

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 our example

- What we should have done:
  - (λx. (λa. x + a) 7) (a + 5)

=α (λx. (λb. x + b) 7) (a + 5)

=β (λb. (a + 5) + b) 7

=β (a + 5) + 7

=β a + 12

The to do at home example

- What you should have done:
  - (λf.(λx. f (f x)) (λy.y+x)

  =α (λf.(λz. f (f z)) (λy.y+x)

  =β λz. (λy.y+x) ((λy.y+x) z)

  =β λz. (λy.y+x) (z+x)

  =β λz. z+x+x

Week 2

- Syntax of λ calculus ✓
- Semantics of λ calculus
  - Informal substitution ✓
  - Free and bound variables ✓
  - Formal substitution ✓
  - Evaluation order

Evaluation order

- What should we reduce first in (λx.x) ((λy.y) z)?
  - A: The inner term: (λy.y) z
  - B: The outer term: (λx.x) ((λy.y) 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 this mean that reduction order doesn’t matter for any program?
  - A: yes B: no

Does evaluation order really not matter?

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

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

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

$=\ \Omega$

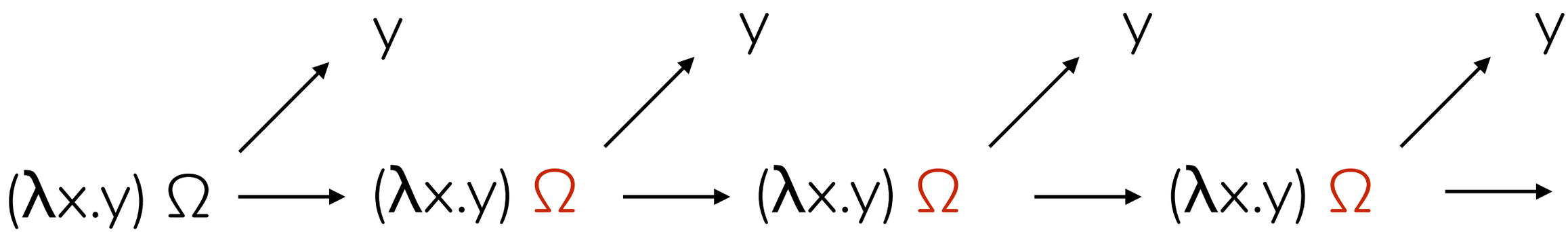
Deja vu!

$\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow$

(Ω has no normal form)

Does evaluation order really not matter?

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



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-name

- Reduce function then apply
  - e_1 e_2
- Haskell's evaluation strategy is call-by-name
  - It only does what is absolutely necessary!
  - Actually it's call-by-need = call-by-name + sharing

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

- Syntax of λ calculus ✓
- Semantics of λ calculus ✓
  - Free and bound variables ✓
  - Substitution ✓
  - Evaluation order ✓

Takeaway

- λ-calculus is a forma system
  - “Simplest reasonable programming language”-Ramsey
  - Binders show up everywhere!
    - Know your capture-avoiding substitution!

Bonus!

Recursive functions in λ-calculus

- Suppose you want to implement factorial in λ-calculus
  - λn. if n ≤ 1 then 1 else n * (fac (n-1))
  - Is this right? A: yes, B: no

The Y combinator

About

Y Combinator created a new model for funding early stage startups.

Twice a year we invest a standard amount of money ($120k) in a large batch of startups.

The startups move to Silicon Valley for 3 months, during which we work intensively with them to get the company into the best possible shape and refine their pitch to investors. Each cycle culminates in Demo Day, when the startups present their companies to a carefully selected, invite-only audience.

But YC doesn't end on Demo Day. We and the YC alumni network continue to help founders for the life of their company, and beyond.

The Y combinator

- Like Ω, there is a special term Y in lambda calculus
  - $Y \stackrel{\text{def}}{=} \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
- Why is this interesting? It reduces as follows
  - $Y\ f$
  - $=_\beta\ f\ (Y\ f)$
  - $=_\beta\ f\ (f\ (Y\ f))$
  - $=_\beta\ f\ (\ldots\ f\ (Y\ f)\ \ldots\ )$

Def: Fixed-point of a function f is a value x such that f x = x

How can we use this?

- Let's go back to factorial example.
  - $f \stackrel{def}{=} \lambda fac.\lambda n.\ \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (fac\ (n-1))$
  - $factorial \stackrel{def}{=} Y\ f$

$=_\beta\ f\ (Y\ f)$

$=\ f\ factorial$

$=\ (\lambda fac.\lambda n.\ \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (fac\ (n-1)))\ factorial$

$=_\beta\ \lambda n.\ \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (factorial\ (n-1))$

How can we use this?

- Apply as usual:

  ➤ factorial 2

  =β (f (Y f)) 2

  =β (λn. if n ≤ 1 then 1 else n * (factorial (n-1))) 2

  =β if 2 ≤ 1 then 1 else 2 * (factorial (2-1))

  =β if 2 ≤ 1 then 1 else 2 * (factorial 1)

  =β 2 * (factorial 1)

How can we use this?

- Apply as usual:

$=_\beta$ 2 * (factorial 1)

$=_\beta$ 2 * ((f factorial) 1)

$=_\beta$ 2 * ((λn. if n ≤ 1 then 1 else n * (factorial (n-1))) 1)

$=_\beta$ 2 * (if 1 ≤ 1 then 1 else 1 * (factorial (1-1)))

$=_\beta$ 2 * (if 1 ≤ 1 then 1 else 1 * (factorial 0))

$=_\beta$ 2 * 1