Fundamentals and lambda calculus

Deian Stefan

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

Logistics

- Assignments:
  - Programming assignment 1 is out
  - Homework 1 will be released tomorrow night
- Podcasting: everything should be set
- Section: 8-8:50AM default; TA choice to do 3-3:50PM
- Clickers, sign up: see piazza/course page for link
  - We'll use them today

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 “fat arrows”
  - Semantics not always the same (`this` has different meaning), but for this class should always be safe to use

fat-arrows.js

In this lecture

In this lecture

In this lecture

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

Before we get started, some terminology

- Syntax (grammar)
  - The symbols used to write a program
  - E.g., `(x + y)` is a grammatical expression
- Semantics
- PL implementation: Syntax -> Semantics

Before we get started, some terminology

- Syntax (grammar)
  - The symbols used to write a program
  - E.g., `(x + y)` is a grammatical expression
- Semantics
  - The actions that occur when a program is executed
- PL implementation: Syntax -> Semantics

Today

- Syntax of λ calculus
- Semantics of λ calculus
  - Free and bound variables
  - 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)
  - 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))

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))`

Example terms

- λx.(2+x)
  - 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))

Example terms

- λx.(2+x)
  - 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))

Example terms

- λx.(2+x)
  - 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:

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:
  - λf.(λx. f (f x))

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:
  - λf.(λx. f (f x))
  - ((λf.(λx. f (f x)) (λx.x+1)) 4

Today

- Syntax of λ calculus ✓
- Semantics of λ calculus
  - Free and bound variables
  - 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!

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))`

Example terms

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

Example terms

- Example: (λx.(2 + x)) 5
  - In JavaScript: `(x => (2 + x)) (5)` → `(2 + 5)` → `7`
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: `(f => (f (3))) (x => (x+1))`

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5)
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: (f => (f (3))) (x => (x+1))

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: (f => (f (3))) (x => (x+1))

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: (f => (f (3))) (x => (x+1))
    → ((x => (x+1)) (3)

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: (f => (f (3))) (x => (x+1))
    → ((x => (x+1)) (3)
    → (3+1) → 4

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  → ((λx.(x + 1)) 3)
  - In JavaScript: (f => (f (3))) (x => (x+1))
    → ((x => (x+1)) (3)
    → (3+1) → 4

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  → ((λx.(x + 1)) 3)
  → (3 + 1) → 4
  - In JavaScript: (f => (f (3))) (x => (x+1))
    → ((x => (x+1)) (3)
    → (3+1) → 4

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

Substitution (not right)

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

A more complicated example

A more complicated example

- Compose function that adds 1 to arg & apply it to 4
  - ((λf.(λx. f (f x)) (λx.x+1)) 4

A more complicated example

- Compose function that adds 1 to arg & apply it to 4
  - ((λf.(λx. f (f x)) (λx.x+1)) 4
  - (λx. (λx.x+1) ((λx.x+1) x)) 4

A more complicated example

- Compose function that adds 1 to arg & apply it to 4
  - ((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)

A more complicated example

- Compose function that adds 1 to arg & apply it to 4
  - ((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)

A more complicated example

- Compose function that adds 1 to arg & apply it to 4
  - ((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

A more complicated example

- Compose function that adds 1 to arg & apply it to 4
  - ((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

Let’s make this even more fun!

Let’s make this even more fun!

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

Let’s make this even more fun!

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

Let's make this even more fun!

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

Let's make this even more fun!

- Instead of 1, let's add x to argument (& do it 2 times):
  - (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)

Let's make this even more fun!

- Instead of 1, let's add x to argument (& do it 2 times):
  - (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

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:
  - $(\lambda f.(\lambda x.\ f\ (f\ x))\ (\lambda y.y+x)$
  - $(\lambda x.\ (\lambda y.y+x)\ ((\lambda y.y+x)\ x)$ the x is **captured**!

Another way to see the problem

- Syntactic sugar: let x = e_1 in $e_2 \stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1$

- Let syntax makes this easy to see:

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

  → let x = a+b in
  let a = 7 in
  (a+b) + a

Another way to see the problem

- Syntactic sugar: let x = e_1 in $e_2 \stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1$

- Let syntax makes this easy to see:

  - let x = a+b in
    let a = 7 in
    x + a
    → let x = a+b in
    let a = 7 in
    (a+b) + a

  - Very obviously wrong!

  - But, guess what: your C macro preprocessor does this!

Fixing the problem

- How can we fix this?

1. Rename variables!

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

2. Do the “dumb” substitution!

Fixing the problem

- How can we fix this?

1. Rename variables!

➤ let x = a+b in let x = a+b in
  let a = 7 in → let a123 = 7 in
  x + a x + a123

2. Do the "dumb" substitution!

Fixing the problem

- How can we fix this?

1. Rename variables!

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

→

let x = a+b in
  let a123 = 7 in
  x + a123

→

let x = a+b in
  let a123 = 7 in
  (a+b) + a123

2. Do the "dumb" substitution!

Why is this the way to go?

- We can always rename bound variables!
  - Def: variable x is bound in λx.(x+y)
- Bound variables are just "placeholders"
  - Above: x is not special, we could have used z
  - We say they are equivalent: λx.(x+y) =α λz.(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:

  - E.g., ∀x. P(x, y) or $\sum_{i \in \{1,\ldots,10\}} x_i + y$

Can we rename everything?

- Can we rename y in $\lambda x.(x+y)$? (A: yes, B: no)
  - No! We don't know what y may be, so we must keep it as is!
- Intuition:
  - E.g., $\forall x.\ P(x, y)$ or $\sum_{i \in \{1,\ldots,10\}} x_i + y$

Can we rename everything?

- Can we rename y in λx.(x+y)? (A: yes, B: no)
  - No! We don't know what y may be, so we must keep it as is!

- Intuition:
  - Can change the name of your function argument variables but not of variables from the outer scope
  - E.g., ∀x. P(x, y) or $\sum_{i \in \{1,\ldots,10\}} x_i + y$

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?

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

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?

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

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?

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

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? No! We say x is bound in λx.(x+y)

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

Def: Capture-avoiding substitution

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

Def: Capture-avoiding substitution

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

Def: Capture-avoiding substitution

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

Def: Capture-avoiding substitution

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

Def: Capture-avoiding substitution

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

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 our example (what we should've done)

Back to our example (what we should've done)

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

Back to our example (what we should've done)

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

  =α (λf.(λz. f (f z)) (λy.y+x)

Back to our example (what we should've done)

- Instead of 1, let's add x to argument (and do it 2x):
  - (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)

Back to our example (what we should've done)

- Instead of 1, let's add x to argument (and do it 2x):
  - (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)

Back to our example (what we should've done)

- Instead of 1, let's add x to argument (and do it 2x):
  - (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

Today

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

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

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 (λx.x) ((λy.y) z)?
  - A: The outer term: (λy.y) z
  - B: The inner term: (λx.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!

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

(Ω has no normal form)

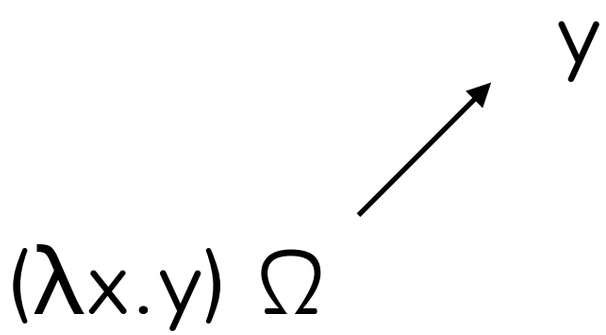
Does evaluation order really not matter?

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

(λx.y) Ω

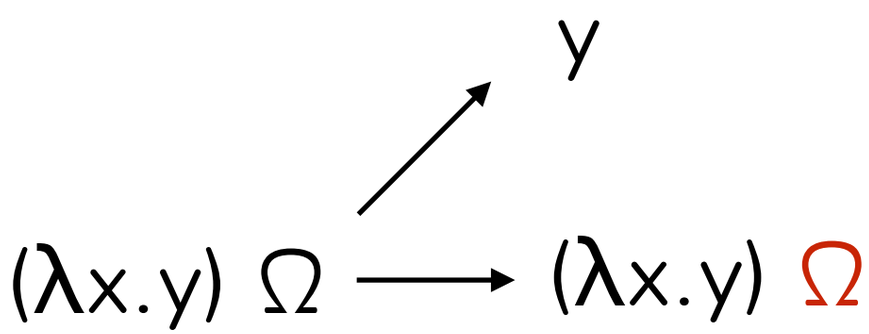
Does evaluation order really not matter?

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



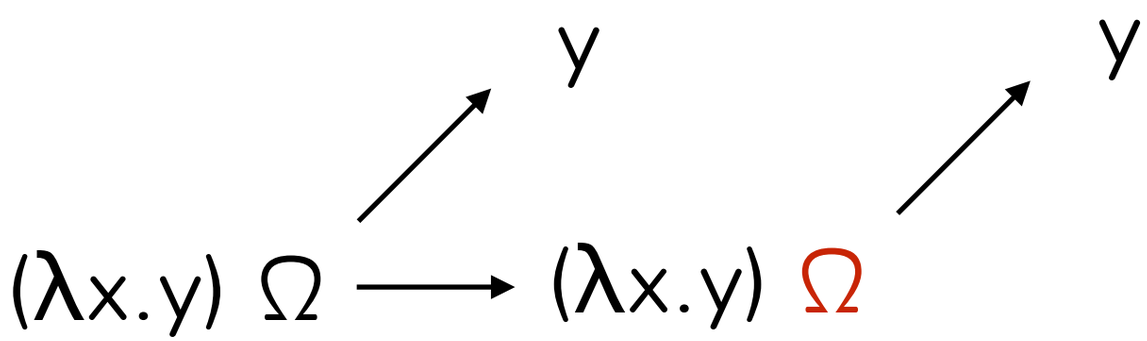
Does evaluation order really not matter?

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



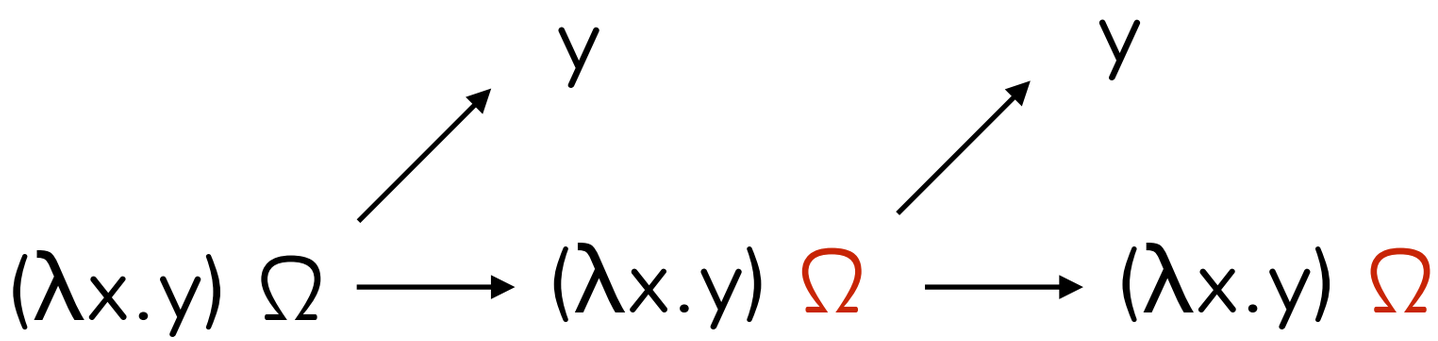
Does evaluation order really not matter?

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



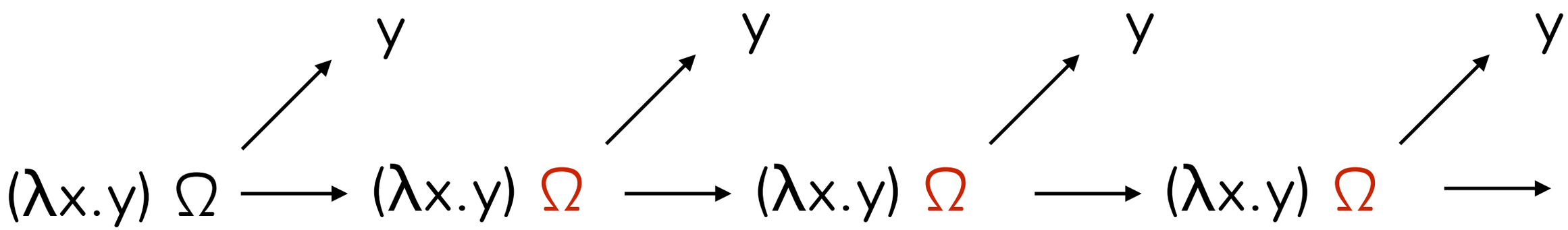
Does evaluation order really not matter?

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



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

Call-by-name

- Reduce function, then reduce args, then apply
  - $e_1\ e_2 \rightarrow \cdots \rightarrow (\lambda x.e_1')\ e_2 \rightarrow e_1'[x:=e_2] \rightarrow \cdots$
- 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

- 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!