Lambda Calculus

Plan

- Introduce lambda calculus
  - Syntax
  - Substitution
  - Operational Semantics (… with contexts!)
  - Evaluation strategies
  - Equality
- Relationship to programming languages (next time)
- Study of types and type systems (later)

#2

Lambda Background

- Developed in 1930's by Alonzo Church
- Subsequently studied by many people (still studied today!)
- Considered the "testbed" for procedural and functional languages
  - Simple
  - Powerful
  - Easy to extend with features of interest
  - Plays similar role for PL research as Turing machines do for computability and complexity
  - Somewhat like a crowbar …

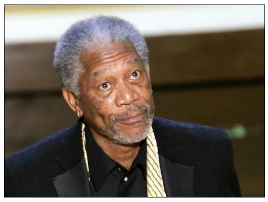
"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus."

(Landin '66)

#3

Lambda Celebrity Representative

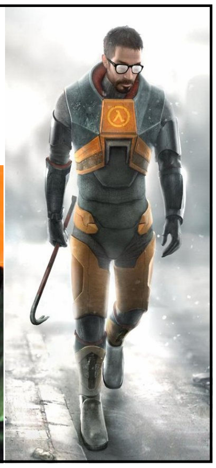
- Milton Friedman?
- Morgan Freeman?
- C. S. Friedman?


#4

Gordon Freeman

- Best-selling PC FPS to date

Lambda Syntax

- The λ-calculus has three kinds of expressions (terms)

$e ::= x$		Variables
	$\mid \lambda x.e$	Functions (abstraction)
	$\mid e_1\ e_2$	Application

- $\lambda x.e$ is a one-argument function with body e
- $e_1\ e_2$ is a function application

- Application associates to the left
  $x\ y\ z$ means $(x\ y)\ z$
- Abstraction extends to the right as far as possible
  $\lambda x.x \lambda y.x\ y\ z$ means $\lambda x.(x\ (\lambda y.\ ((x\ y)\ z)))$

#6

Why Should I Care?

- A language with 3 expressions? Woof!
- Li and Zdancewick. *Downgrading policies and relaxed noninterference*. POPL ’05
  - **Just one example** of a recent PL/security paper …

4. LOCAL DOWNGRADING POLICIES

4.1 Label Definition

Definition 4.1.1 (The policy language). *In Figure 1.*

Types	$\tau ::=$	$\mathsf{int} \mid \tau \to \tau$
Constants	$c ::=$	c_i
Operators	$\oplus ::=$	$+, -, =, \ldots$
Terms	$m ::=$	$\lambda x\!:\!\tau.\ m \mid m\ m \mid x \mid c \mid m \oplus m$
Policies	$n ::=$	$\lambda x\!:\!\mathsf{int}.\ m$
Labels	$l ::=$	$\{n_1, \ldots, n_k\} \quad (k \geq 1)$

Figure 1: $\mathbb{L}_{\mathsf{local}}$ Label Syntax

The core of the policy language is a variant of the simply-typed λ-calculus with a base type, binary operators and constants. A **downgrading policy** is a λ-term that specifies how an integer can be downgraded: when this λ-term is applied to the annotated integer, the result becomes public. A

$$\frac{\Gamma \vdash m : \tau}{\Gamma \vdash m \equiv m : \tau}\ \textsc{Q-Refl}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau}{\Gamma \vdash m_2 \equiv m_1 : \tau}\ \textsc{Q-Symm}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau \qquad \Gamma \vdash m_2 \equiv m_3 : \tau}{\Gamma \vdash m_1 \equiv m_3 : \tau}\ \textsc{Q-Trans}$$

$$\frac{\Gamma, x\!:\!\tau_1 \vdash m_1 \equiv m_2 : \tau_2}{\Gamma \vdash \lambda x\!:\!\tau_1.\ m_1 \equiv \lambda x\!:\!\tau_1.\ m_2 : \tau_1 \to \tau_2}\ \textsc{Q-Abs}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau_1 \to \tau_2 \qquad \Gamma \vdash m_3 \equiv m_4 : \tau_1}{\Gamma \vdash m_1\ m_3 \equiv m_2\ m_4 : \tau_2}\ \textsc{Q-App}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \mathsf{int} \qquad \Gamma \vdash m_3 \equiv m_4 : \mathsf{int}}{\Gamma \vdash m_1 \oplus m_3 \equiv m_2 \oplus m_4 : \mathsf{int}}\ \textsc{Q-BinOp}$$

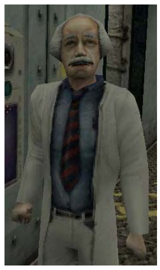
Examples of Lambda Expressions

- The identity function:

  I $=_{def}$ λx. x
- A function that given an argument y discards it and yields the identity function:

  λy. (λx. x)
- A function that given a function f invokes it on the identity function

  λf. f (λx. x)



“There goes our grant money.”

#8

Scope of Variables

- As in all languages with variables it is important to discuss the notion of scope
  - The scope of an identifier is the portion of a program where the identifier is accessible
- An abstraction λx. E binds variable x in E
  - x is the newly introduced variable
  - E is the scope of x (unless x is shadowed …)
  - We say x is bound in λx. E
  - Just like formal function arguments are bound in the function body

#9

Free and Bound Variables

- A variable is said to be free in E if it has occurrences that are not bound in E
- We can define the free variables of an expression E recursively as follows:

  Free(x) = { x }

  Free(E_1 E_2) = Free(E_1) ∪ Free(E_2)

  Free(λx. E) = Free(E) - { x }
- Example: Free(λx. x (λy. x y z)) = { z }
- Free variables are (implicitly or explicitly) declared outside the term

#10

Free Your Mind!

- Just like in any language with statically nested scoping we have to worry about variable shadowing
  - An occurrence of a variable might refer to different things in different contexts
- e.g., IMP with locals: let x = E in x + (let x = E’ in x) + x
- In λ-calculus: λx. x (λx. x) x

#11

Renaming Bound Variables

- λ-terms that can be obtained from one another by renaming of the bound variables are considered identical.
- This is called α-equivalence.
- Renaming bound vars is called α-renaming.
- Example: λx. x is identical to λy. y and to λz. z
- Intuition:
  - By changing the name of a formal argument and of all its occurrences in the function body, the behavior of the function *does not change*
  - In λ-calculus such functions are considered identical

#12

Make It Easy On Yourself

- Convention: we will always try to rename bound variables so that they are all unique
  - e.g., write $\lambda x.\ x\ (\lambda y.y)\ x$ instead of $\lambda x.\ x\ (\lambda x.x)\ x$
- This makes it easy to see the scope of bindings and also prevents confusion!

Substitution

- The substitution of E' for x in E (written [E'/x]E)
  - Step 1. Rename bound variables in E and E' so they are unique
  - Step 2. Perform the textual substitution of E' for x in E
- Called capture-avoiding substitution.
- Example: $[y\ (\lambda x.\ x)\ /\ x]\ \lambda y.\ (\lambda x.\ x)\ y\ x$
  - After renaming: $[y\ (\lambda v.\ v)/x]\ \lambda z.\ (\lambda u.\ u)\ z\ x$
  - After substitution: $\lambda z.\ (\lambda u.\ u)\ z\ (y\ (\lambda v.\ v))$
- If we are not careful with scopes we might get:

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

#14

The deBruijn Notation

- An alternative syntax that avoids naming of bound variables (and the subsequent confusions)
- The deBruijn index of a variable *occurrence* is the number of lambdas that separate the occurrence from its binding lambda in the abstract syntax tree
- The deBruijn notation replaces names of occurrences with their deBuijn index
- Examples:
  - $\lambda x.x$ — $\lambda.0$
  - $\lambda x.\lambda x.x$ — $\lambda.\lambda.0$
  - $\lambda x.\lambda y.y$ — $\lambda.\lambda.0$
  - $(\lambda\ x.\ x\ x)\ (\lambda\ z.\ z\ z)$ — $(\lambda.0\ 0)\ (\lambda.0\ 0)$
  - $\lambda x.\ (\lambda x.\ \lambda y.x)\ x$ — $\lambda.(\lambda.\lambda.1)\ 0$

Identical terms have identical representations !

#15

Combinators

- A λ-term without free variables is closed or a combinator
- Some interesting combinators:

  I = $\lambda x.\ x$
  K = $\lambda x.\lambda y.x$
  S = $\lambda f.\lambda g.\lambda x.f\ x\ (g\ x)$
  D = $\lambda x.\ x\ x$
  Y = $\lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
- Theorem: Any closed term is equivalent to one written with just S, K, I
  - Example: $D =_\beta S\ I\ I$
  - (we'll discuss this form of equivalence in a bit)

#16

Informal Semantics

- We consider only closed terms
- The evaluation of

  $(\lambda x.\ e)\ e'$

  1. Binds x to e'
  2. Evaluates e with the new binding
  3. Yields the result of this evaluation
- Like a function call, or like "let x = e' in e"
- Example:

  $(\lambda f.\ f\ (f\ e))\ g$ evaluates to $g\ (g\ e)$

#17

Operational Semantics

- Many operational semantics for the λ-calculus
- All are based on the equation

  $(\lambda x.\ e)\ e' =_\beta [e'/x]e$

  usually read from left to right
- This is called the β-rule and the evaluation step a β-reduction
- The subterm $(\lambda x.\ e)\ e'$ is a β-redex
- We write $e \to_\beta e'$ to say that e β-reduces to e' in one step
- We write $e \to_\beta^* e'$ to say that e β-reduces to e' in 0 or more steps
  - Should remind you of small-step opsem term-rewriting

#18

Examples of Evaluation

- The identity function:
  $(\lambda x.\ x)\ E \rightarrow [E\ /\ x]\ x = E$
- Another example with the identity:
  $(\lambda f.\ f\ (\lambda x.\ x))\ (\lambda x.\ x) \rightarrow$
  $[\lambda x.\ x\ /\ f]\ f\ (\lambda x.\ x)) = [(\lambda x.\ x)\ /\ f]\ f\ (\lambda y.\ y)) =$
  $(\lambda x.\ x)\ (\lambda y.\ y) \rightarrow$
  $[\lambda y.\ y\ /x]\ x = \lambda y.\ y$
- A non-terminating evaluation:
  $(\lambda x.\ xx)(\lambda y.\ yy) \rightarrow$
  $[\lambda y.\ yy\ /\ x]xx = (\lambda y.\ yy)(\lambda y.\ yy) \rightarrow\ \ldots$
- Try T T, where $T = \lambda x.x\ x\ x$

Evaluation and the Static Scope

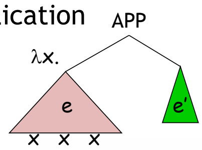
- The definition of substitution guarantees that evaluation respects static scoping:
  $(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \rightarrow_\beta \lambda z.\ z\ (y\ (\lambda v.\ v))$

  (y remains free, i.e., defined externally)
- If we forget to rename the bound y:
  $(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \rightarrow^*_\beta \lambda y.\ y\ (y\ (\lambda v.\ v))$

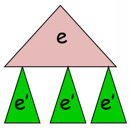
  (y was free before but is bound now)

Another View of Reduction

- The application



- becomes:



Terms can "grow" substantially through β-reduction !

Normal Forms

- A term without redexes is in normal form
- A reduction sequence stops at a normal form
- If e is in normal form and $e \rightarrow^*_\beta e'$ then e is identical to e'
- $K = \lambda x.\lambda y.\ x$ is in normal form
- K I is *not* in normal form

Nondeterministic Evaluation

- We define a small-step reduction relation

$$\frac{}{(\lambda x.\ e)\ e' \rightarrow [e'/x]e}$$

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{e_1\ e_2 \rightarrow e_1\ e_2'}$$

$$\frac{e \rightarrow e'}{\lambda x.\ e\ \rightarrow \lambda x.\ e'}$$

- This is a non-deterministic semantics
- Note that we evaluate under λ *(where?)*

Lambda Calculus Contexts

- Define contexts with one hole
  $H ::= \bullet\ |\ \lambda x.\ H\ |\ H\ e\ |\ e\ H$
- Write H[e] to denote the filling of the hole in H with the expression e
- Example:
  $H = \lambda x.\ x\ \bullet \qquad H[\lambda y.y] = \lambda x.\ x\ (\lambda y.\ y)$
- Filling the hole allows variable capture!
  $H = \lambda x.\ x\ \bullet \qquad H[x] = \lambda x.x\ x$

Contextual Opsem

	$e \rightarrow e'$
$(\lambda x.\ e)\ e' \rightarrow [e'/x]e$	$H[e] \rightarrow H[e']$

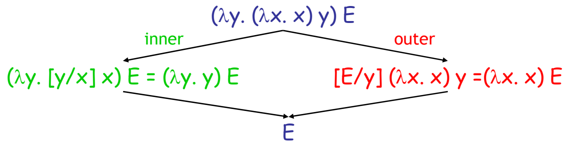
- Contexts allow concise formulations of congruence rules (application of local reduction rules on subterms)
- Reduction occurs at a β-redex that can be anywhere inside the expression
- The latter rule is called a congruence or structural rule
- The above rules do not specify which redex must be reduced first

The Order of Evaluation

- In a λ-term there could be more than one instance of $(\lambda x.\ E)\ E'$, as in:

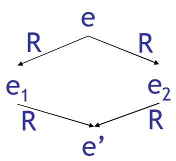
  $(\lambda y.\ (\lambda x.\ x)\ y)\ E$

  - could reduce the inner or the outer λ
  - which one should we pick?



The Diamond Property

- A relation R has the diamond property if whenever $e\ R\ e_1$ and $e\ R\ e_2$ then there exists e' such that $e_1\ R\ e'$ and $e_2\ R\ e'$



- $\rightarrow_\beta$ does *not* have the diamond property
- $\rightarrow_\beta^*$ has the diamond property
- Also called the confluence property

The Diamond Property

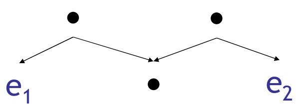
- Languages defined by non-deterministic sets of rules are common
  - Logic programming languages
  - Expert systems
  - Constraint satisfaction systems
    - and thus most pointer analyses …
  - Dataflow systems
  - Makefiles
- It is useful to know whether such systems have the diamond property

(Beta) Equality

- Let $=_\beta$ be the reflexive, transitive and symmetric closure of $\rightarrow_\beta$

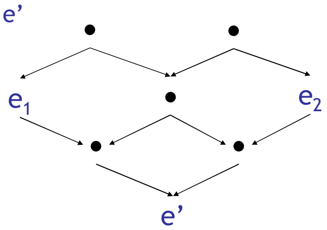
  $=_\beta$ is $(\rightarrow_\beta \cup \leftarrow_\beta)^*$

- That is, $e_1 =_\beta e_2$ if e_1 converts to e_2 via a sequence of forward and backward $\rightarrow_\beta$



The Church-Rosser Theorem

- If $e_1 =_\beta e_2$ then there exists e' such that $e_1 \rightarrow_\beta^* e'$ and $e_2 \rightarrow_\beta^* e'$



- Proof (informal): apply the diamond property as many times as necessary

Corollaries

- If $e_1 =_\beta e_2$ and e_1 and e_2 are normal forms then e_1 is identical to e_2
  - From C-R we have $\exists e'.\ e_1 \to^*_\beta e'$ and $e_2 \to^*_\beta e'$
  - Since e_1 and e_2 are normal forms they are identical to e'
- If $e \to^*_\beta e_1$ and $e \to^*_\beta e_2$ and e_1 and e_2 are normal forms then e_1 is identical to e_2
  - "All terms have a unique normal form."

#31

Evaluation Strategies

- Church-Rosser theorem says that independent of the reduction strategy we will find $\leq$ 1 normal form
- But some reduction strategies might find 0
  - $(\lambda x.\ z)\ \underline{((\lambda y.y\ y)\ (\lambda y.y\ y))} \to (\lambda x.\ z)\ \underline{((\lambda y.y\ y)\ (\lambda y.y\ y))} \to$ …
  - $\underline{(\lambda x.\ z)\ ((\lambda y.y\ y)\ (\lambda y.y\ y))} \to z$
- There are three traditional strategies
  - normal order (never used, always works)
  - call-by-name (rarely used, cf. TeX)
  - call-by-value (amazingly popular)

#32

Call To Power (By Value)

- Normal Order
  - Evaluate the left-most redex not contained in another redex
  - If there is a normal form, this finds it
  - Not used in practice: requires partially evaluating function pointers and looking "inside" functions
- Call-By-Name ("lazy")
  - Don't reduce under λ, don't evaluate a function argument (until you need to)
  - Does not always evaluate to a normal form
- Call-By-Value ("strict" or "eager")
  - Don't reduce under λ, *do* evaluate a function's argument right away
  - Finds normal forms less often than the other two

#33

Endgame

- This time: λ syntax, semantics, reductions, equality, …
- Next time: encodings, real programs, type systems, and all the fun stuff!

"Wisely done, Mr. Freeman. I will see you up ahead."

Homework

- Project Proposal Due Two Days Ago …
- Read Leroy article, think about axiomatic
- No Class Tuesday (projects, HW5)
- Homework 5 Due Next Thursday