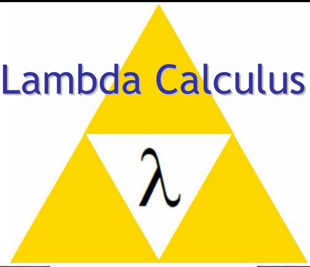
Lambda Calculus



Plan

- Introduce lambda calculus
  - Syntax
  - Substitution
  - Operational Semantics (… with contexts!)
  - Evaluations strategies
  - Equality
- Later:
  - Relationship to programming languages
  - Study of types and type systems

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 new features of interest
  - Lambda:PL :: Turning Machine:Complexity
  - Somewhat like a crowbar …

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

(Landin '66)

Lambda Syntax

- The λ-calculus has 3 kinds of expressions (terms)

e ::= x		Variables
|	λx. e	Functions (abstractions)
|	e_1 e_2	Application

- λx. e is a one-argument anonymous function with body e
- e_1 e_2 is a function application
- Application associates to the left

  x y z === (x y) z
- Abstraction extends far to the right

  λx. x λy. x y z === λx. (x [λy. {(x y) z}])

Why Should I Care?

- A language with 3 expressions? Woof!
- Li and Zdancewic. *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	τ ::=	int | $\tau \to \tau$
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 ::=	$\{n_1, \ldots, n_k\}$ $(k \geq 1)$

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

$$\frac{\Gamma \vdash m : \tau}{\Gamma \vdash m \equiv m : \tau}\ \text{Q-REFL}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau}{\Gamma \vdash m_2 \equiv m_1 : \tau}\ \text{Q-SYMM}$$

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

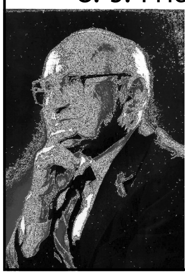
$$\frac{\Gamma, x{:}\tau_1 \vdash m_1 \equiv m_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ m_1 \equiv \lambda x{:}\tau_1.\ m_2 : \tau_1 \to \tau_2}\ \text{Q-ABS}$$

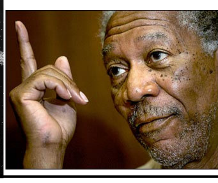
$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau_1 \to \tau_2 \quad \Gamma \vdash m_3 \equiv m_4 : \tau_1}{\Gamma \vdash m_1\ m_3 \equiv m_2\ m_4 : \tau_2}\ \text{Q-APP}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \mathsf{int} \quad \Gamma \vdash m_3 \equiv m_4 : \mathsf{int}}{\Gamma \vdash m_1 \oplus m_3 \equiv m_2 \oplus m_4 : \mathsf{int}}\ \text{Q-BINOP}$$

Lambda Celebrity Representative

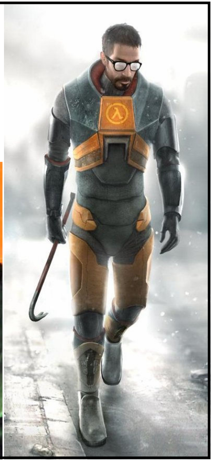
- Milton Friedman?
- Morgan Freeman?
- C. S. Friedman?





Gordon Freeman

- Best-selling PC FPS to date …



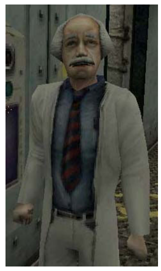
Examples of Lambda Expressions

- 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 an 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}
  - $\text{Free}(E_1\ E_2) = \text{Free}(E_1) \cup \text{Free}(E_2)$
  - Free(λx. E) = Free(E) - {x}
- Example: Free(λx. x (λy. x y z)) = {z}
- Free variables are (implicitly or explicitly) declared outside the expression

Free Your Mind!

- Just as 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
- Example in IMP with locals:

  let x = 5 in x + (let x = 9 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 bound variables are considered *identical*
- This is called α-equivalence
- Renaming bound vars is called α-renaming
- Ex: λx. x is identical to λy. y and to λz. z
- Intuition:
  - By changing the name of a formal argument and all of 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 λx. x (λy.y) x instead of λx. x (λx.x) x
- This makes it easy to see the scope of bindings and also prevents confusion!

Substitution

- The substitution of F for x in E (written [F/x]E)
  - Step 1. Rename bound variables in E and F so they are unique
  - Step 2. Perform the textual substitution of f for X in E
- Called capture-avoiding substitution
- Example: [y (λx. x) / x] λy. (λx. x) y x
  - After renaming: [y (λx. x) / x] λz. (λu. u) z x
  - After substitution: λz. (λu. u) z (y (λx. x))
- If we are not careful with scopes we might get:

  λy. (λx. x) y (y (λx. x)) ← wrong!

#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 that number of lambda that separate the occurrence from its binding lambda in the abstract syntax tree
- The deBruijn notation replaces names of occurrences with their deBruijn indices
- Examples:

– **λ x. x**	**λ. 0**	Identical terms have identical representations!
– **λ x. λ x. x**	**λ. λ. 0**	
– **λ x. λ y. y**	**λ. λ. 0**	
– **(λ x. x x) (λ z. z z)**	**(λ. 0 0) (λ. 0 0)**	
– **λ x. (λ x. λ y. x) x**	**λ. (λ. λ. 1) 0**	

#15

Combinators

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

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

#16

Informal Semantics

- We consider only closed terms
- The evaluation of

  (λ x. e) f
  - Binds x to f
  - Evaluates e *with the new binding*
  - Yields the result of this evaluation
- Like a function call, or like "let x = f in e"
- Example:

  (λ 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

  (λ x. e) f $=_\beta$ [f/x]e

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

#18

Examples of Evaluation

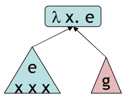
- The identity function:
  $(\lambda\, x.\, x)\ E \to [E / x]\ x = E$
- Another example with the identity:
  $(\lambda\, f.\, f\ (\lambda\, x.\, x))\ (\lambda\, x.\, x) \to$
  $[\lambda\, x.\, x / f]\ f\ (\lambda\, x.\, x)) =$
  $[\lambda\, x.\, x / f]\ f\ (\lambda\, y.\, y)) =$
  $(\lambda\, x.\, x)\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

Evaluation and the Static Scope

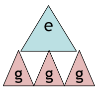
- The definition of substitution guarantees that evaluation respets static scoping:
  $(\lambda\ x.\ (\lambda\ y.\ y\ x))\ (y\ (\lambda\ x.\ x)) \to_\beta \lambda\ z.\ z\ (y\ (\lambda\ v.\ v))$
  (y remains free, i.e., defined externally)
- If we forget to rename the bound y:
  $(\lambda\ x.\ (\lambda\ y.\ y\ x))\ (y\ (\lambda\ x.\ x)) \to_\beta^{*} \lambda\ y.\ y\ (y\ (\lambda\ v.\ v))$
  (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 $e \to_\beta^{*} f$ then e is identical to f
- $K = \lambda\ x.\ \lambda\ y.\ x$ is in normal form
- K I is *not* in normal form

Nondeterministic Evaluation

- We define a small-step reduction relation

$$\frac{}{(\lambda\ x.\ e)\ f \to [f/x]e}$$

$$\frac{e_1 \to e_2}{e_1\ f \to e_2\ f} \qquad \frac{f_1 \to f_2}{e\ f_1 \to e\ f_2}$$

$$\frac{e \to f}{\lambda\ x.\ e \to \lambda\ x.\ f}$$

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

Lambda Calculus Contexts

- Define contexts with one hole
- $H ::= \bullet \mid \lambda\ x.\ H \mid H\ e \mid e\ H$
- 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

$$\frac{}{(\lambda\ x.\ e)\ f \to [f/x]e} \qquad \frac{e \to f}{H[e] \to H[f]}$$

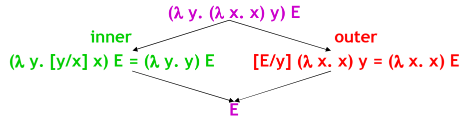
- Contexts allow concise formulations of congruence rules (application of local reduction rules on subterms)
- Reduction occurs at a β-redex that can be anywhere inside the expression
- The latter rule is called a congruence or structural rule
- The above rules to not specify which redex must be reduced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

The Order of Evaluation

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

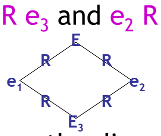
$$(\lambda\ y.\ (\lambda\ x.\ x)\ y)\ E$$

  - Could reduce the inner or 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_3 such that $e_1\ R\ e_3$ and $e_2\ R\ e_3$



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

A Diamond In The Rough

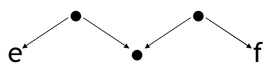
- Languages defined by non-deterministic sets of rules are common
  - Logic programming languages
  - Expert systems
  - Constraint satisfaction systems
    - And thus most pointer analyses …
  - Dataflow systems
  - Makefiles
- It is useful to know whether such systems have the diamond property

(Beta) Equality

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

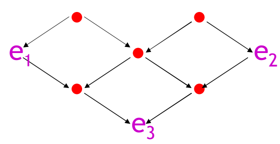
$$=_\beta \textbf{ is } (\to_\beta \cup \leftarrow_\beta)^*$$

- That is, $e =_\beta f$ if e converts to f via a sequence of forward and backward $\to_\beta$



The Church-Rosser Theorem

- If $e_1 =_\beta e_2$ then there exists e_3 such that $e_1 \to_\beta{}^* e_3$ and $e_2 \to_\beta{}^* e_3$



- 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 $\exists e_3.\ e_1 \to_\beta^* e_3$ and $e_2 \to_\beta^* e_3$
  - Since e_1 and e_2 are normal forms they are identical to e_3
- 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) $\underline{((\lambda \text{ y. y y}) (\lambda \text{ y. y y}))} \to$
  $(\lambda$ x. z) $\underline{((\lambda \text{ y. y y}) (\lambda \text{ y. y y}))} \to \ldots$
- $\underline{(\lambda \text{ x. z}) ((\lambda \text{ y. y y}) (\lambda \text{ y. y y}))} \to$ 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

Civilization: Call By Value

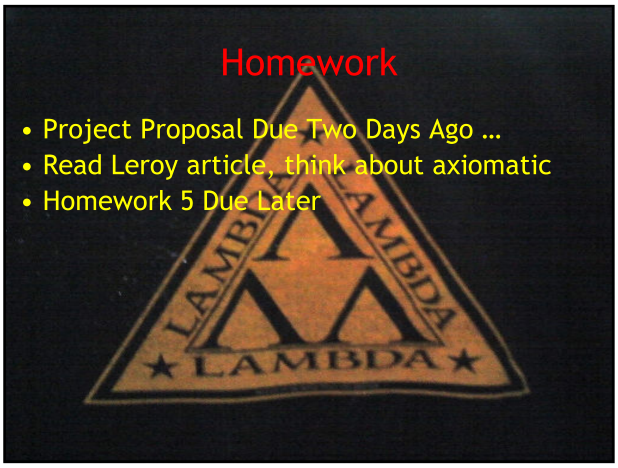
- Normal Order
  - Evaluates the left-most redex not contained in another redex
  - If there is a normal form, this finds it
  - Not used in practice: requires partially evaluating function pointers and looking "inside" functions
- Call-By-Name ("lazy")
  - Don't reduce under λ, don't evaluate a function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

Endgame

- This time: λ syntax, semantics, reductions, equality, …
- Next time: encodings, real prorams, 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
- Homework 5 Due Later

Tricksy On The Board Answer

- Is this rule unsound?

$$\frac{\vdash \{A \wedge p\}\ c_{then}\ \{B_{then}\} \qquad \vdash \{A \wedge \neg p\}\ c_{else}\ \{B_{else}\}}{\vdash \{A\}\ \text{if } p \text{ then } c_{then} \text{ else } c_{else}\ \{B_{then} \vee B_{else}\}}$$

- Nope: it's our basic rule plus 2x consequence

$$\frac{\vdash \{A \wedge p\}\ c_1\ \{B\} \qquad \vdash \{A \wedge \neg p\}\ c_2\ \{B\}}{\vdash \{A\}\ \text{if } p \text{ then } c_1 \text{ else } c_2\ \{B\}}$$

$$\frac{\vdash A' \Rightarrow A \qquad \vdash \{A\}\ c\ \{B\} \qquad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c\ \{B'\}}$$

- Note that $B_{then} \Rightarrow B_{then} \vee B_{else}$

#36