

CMSC 631 – Program Analysis and Understanding Fall 2003

Lambda Calculus and Functional Programming

Space of Program Analyses

Data flow analysis

Type systems

Theorem proving

Model checking

Motivation

- Commonly-used programming languages are large and complex
 - ANSI C99 standard: 538 pages
 - ANSI C++ standard: 714 pages
 - Java language specification 2.0: 505 pages
- Not good vehicles for understanding language features or explaining program analysis

Goal

- Develop a “core language” that has
 - The essential features
 - No overlapping constructs
 - And none of the cruft
 - Extra features of full language can be defined in terms of the core language (“syntactic sugar”)
- Lambda calculus
 - Standard core language for single-threaded procedural programming
 - Often with added features (e.g., state); we’ll see that later

Lambda Calculus

- Syntax:

$e ::= x$	variable
$\lambda x.e$	function abstraction
$e e$	function application

- Only constructs in pure lambda calculus
 - Functions take functions as arguments and return functions as results
 - I.e., the lambda calculus supports *higher-order functions*

Semantics

- To evaluate $(\lambda x.e1) e2$
 - Bind x to $e2$
 - Evaluate $e1$
 - Return the result of the evaluation
- This is called “beta-reduction”
 - $(\lambda x.e1) e2 \rightarrow_{\beta} e1[e2/x]$
 - $(\lambda x.e1) e2$ is called a *redex*
 - We’ll usually omit the beta

Three Conveniences

- Syntactic sugar for local declarations
 - $\text{let } x = e1 \text{ in } e2$ is short for $(\lambda x.e2) e1$
- Scope of λ extends as far to the right as possible
 - $\lambda x.\lambda y.x y$ is $\lambda x.(\lambda y.(x y))$
- Function application is left-associative
 - $x y z$ is $(x y) z$

Scoping and Parameter Passing

- Beta-reduction is not yet precise
 - $(\lambda x.e1) e2 \rightarrow e1[e2/x]$
 - what if there are multiple x 's?
- Example:
 - $\text{let } x = a \text{ in let } y = \lambda z.x \text{ in let } x = b \text{ in } y x$
 - which x 's are bound to a , and which to b ?

Static (Lexical) Scope

- Just like most languages, a variable refers to the closest definition
- Make this precise using variable renaming
 - The term
 - $\text{let } x = a \text{ in let } y = \lambda z.x \text{ in let } x = b \text{ in } y x$
 - is "the same" as
 - $\text{let } x = a \text{ in let } y = \lambda z.x \text{ in let } w = b \text{ in } y w$
 - Variable names don't matter

Free Variables and Alpha Conversion

- The set of *free variables* of a term is

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.e) &= FV(e) - \{x\} \\ FV(e1 e2) &= FV(e1) \cup FV(e2) \end{aligned}$$

- A term e is *closed* if $FV(e) = \emptyset$
- A variable that is not free is *bound*

Alpha Conversion

- Terms are equivalent up to renaming of bound variables
 - $\lambda x.e = \lambda y.(e[y/x])$ if $y \notin FV(e)$
- This is often called *alpha conversion*, and we will use it implicitly whenever we need to avoid capturing variables when we perform substitution

Substitution

- Formal definition:
 - $x[e/x] = e$
 - $z[e/x] = z$ if $z \neq x$
 - $(e1 e2)[e/x] = (e1[e/x] e2[e/x])$
 - $(\lambda z.e1)[e/x] = \lambda z.(e1[e/x])$ if $z \neq x$ and $z \notin FV(e)$
- Example:
 - $(\lambda x.y x) x =_{\alpha} (\lambda w.y w) x \rightarrow_{\beta} y x$
 - (We won't write alpha conversion down in the future)

A Note on Substitutions

- People write substitution many different ways
 - $e1[e2/x]$
 - $e1[x \mapsto e2]$
 - $[x/e2]e1$
 - and more...
 -
- But they all mean the same thing

Multi-Argument Functions

- We can't (yet) write multi-argument functions
 - E.g., a function of two arguments $\lambda(x, y).e$
- Trick: Take arguments one at a time
 - $\lambda x.\lambda y.e$
 - This is a function that, given argument x , returns a function that, given argument y , returns e
 - $(\lambda x.\lambda y.e) a b \rightarrow (\lambda y.e[a/x]) b \rightarrow e[a/x][b/y]$
- This is often called *Currying* and can be used to represent functions with any # of arguments

Booleans

- $true = \lambda x.\lambda y.x$
- $false = \lambda x.\lambda y.y$
- $\text{if } a \text{ then } b \text{ else } c = a b c$
- Example:
 - $\text{if } true \text{ then } b \text{ else } c \rightarrow (\lambda x.\lambda y.x) b c \rightarrow (\lambda y.b) c \rightarrow b$
 - $\text{if } false \text{ then } b \text{ else } c \rightarrow (\lambda x.\lambda y.y) b c \rightarrow (\lambda y.y) c \rightarrow c$

Combinators

- Any closed term is also called a *combinator*
 - So $true$ and $false$ are both combinators
- Other popular combinators
 - $I = \lambda x.x$
 - $S = \lambda x.\lambda y.x$
 - $K = \lambda x.\lambda y.\lambda z.x z (y z)$
 - Can also define calculi in terms of combinators
 - E.g., the SKI calculus
 - Turns out the SKI calculus is also Turing complete

Pairs

- $(a, b) = \lambda x.\text{if } x \text{ then } a \text{ else } b$
- $\text{fst} = \lambda p.p \text{ true}$
- $\text{snd} = \lambda p.p \text{ false}$
- Then
 - $\text{fst } (a, b) \rightarrow^* a$
 - $\text{snd } (a, b) \rightarrow^* b$

Natural Numbers (Church)

- $0 = \lambda x.\lambda y.y$
- $1 = \lambda x.\lambda y.x y$
- $2 = \lambda x.\lambda y.x(x y)$
- i.e., $n = \lambda x.\lambda y.<\text{apply } x \text{ n times to } y>$
- $\text{succ} = \lambda z.\lambda x.\lambda y.x(z x y)$
- $\text{iszero} = \lambda z.z (\lambda y.\text{false}) \text{ true}$

Natural Numbers (Scott)

- $0 = \lambda x. \lambda y. x$
- $1 = \lambda x. \lambda y. y$
- $2 = \lambda x. \lambda y. y \ 1$
- I.e., $n = \lambda x. \lambda y. y \ (n-1)$
-
- $\text{succ} = \lambda z. \lambda x. \lambda y. y \ z$
- $\text{pred} = \lambda z. z \ 0 \ (\lambda x. x)$
- $\text{iszero} = \lambda z. z \ \text{true} \ (\lambda x. \text{false})$

Operational Semantics

- An operational semantics is a series of rules for evaluating (“running”) a program
 - Example: Eval() from last time
- So far we’ve defined one operational semantic rule, but it’s still not precise
 - $(\lambda x. e1) \ e2 \rightarrow e1[e2/x]$
 - Where does this rule apply?
 - Current answer: Anywhere within a term

A Nondeterministic Semantics

$$\frac{}{(\lambda x. e1) \ e2 \rightarrow e1[e2/x]} \quad \frac{e \rightarrow e'}{(\lambda x. e) \rightarrow (\lambda x. e')}$$

$$\frac{e1 \rightarrow e1'}{e1 \ e2 \rightarrow e1' \ e2} \quad \frac{e2 \rightarrow e2'}{e1 \ e2 \rightarrow e1 \ e2'}$$

Natural Deduction

- These are *natural deduction* style rules

$$\frac{H1 \ H2 \ \dots \ Hn}{C}$$
 - Read: If hypotheses $H1$ through Hn hold, then conclusion C holds
 -
- The rules are axioms that define something, in this case what \rightarrow means
-
- We will use this style of rule extensively

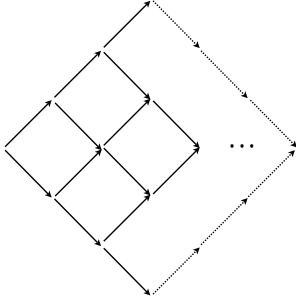
Example

- We can apply reduction anywhere in a term
 - $(\lambda x. (\lambda y. y) \ x) \ ((\lambda z. w) \ x) \rightarrow \lambda x. (x \ ((\lambda z. w) \ x)) \rightarrow \lambda x. x \ w$
 - $(\lambda x. (\lambda y. y) \ x) \ ((\lambda z. w) \ x) \rightarrow \lambda x. (\lambda y. y \ x) \ (w) \rightarrow \lambda x. x \ w$
 -
- Does the order of evaluation matter?

The Church-Rosser Theorem

- Lemma (The Diamond Property):
 - If $a \rightarrow^* b$ and $a \rightarrow^* c$, there there exists d such that $b \rightarrow^* d$ and $c \rightarrow^* d$
- Church-Rosser Theorem:
 - If $a \rightarrow^* b$ and $a \rightarrow^* c$, there there exists d such that $b \rightarrow^* d$ and $c \rightarrow^* d$
- Proof: By diamond property

Proof



Normal Form

- A term is in *normal form* if it cannot be reduced
 - Examples: $\lambda x.x$, $\lambda x.\lambda y.z$
- By Church-Rosser Theorem, every term reduces to at most one normal form
 - Warning: All of this applies only to the pure lambda calculus with non-deterministic evaluation

Beta-Equivalence

- Let $=_{\beta}$ be the reflexive, symmetric, and transitive closure of \rightarrow
 - E.g., $(\lambda x.x) y \rightarrow y \leftarrow (\lambda z.\lambda w.z) y y$, so all three are beta equivalent
- If $a =_{\beta} b$, then there exists c such that $a \rightarrow^* c$ and $b \rightarrow^* c$
 - Proof: Consequence of Church-Rosser Theorem
- In particular, if $a =_{\beta} b$ and both are normal forms, then they are equal

Not Every Term Has a Normal Form

- Consider
 - $\Delta = \lambda x.x x$
 - Then $\Delta \Delta \rightarrow \Delta \Delta \rightarrow \dots$
- In general, *self application* leads to loops
 - ...which is good if we want recursion

A Fixpoint Combinator

- Also called a paradoxical combinator
 - $Y = \lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))$
 - Note: There are many versions of this combinator
- Then $Y F =_{\beta} F (Y F)$
 - $Y F = (\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) F$
 - $\rightarrow (\lambda x.F (x x)) (\lambda x.F (x x))$
 - $\rightarrow F ((\lambda x.F (x x)) (\lambda x.F (x x)))$
 - $\leftarrow F (Y F)$

Example

- $\text{Fact } n = \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}(n-1)$
- Let $G = \lambda f.<\text{body of factorial}>$
 - I.e., $G = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$
- $Y G I =_{\beta} G (Y G) I$
 - $=_{\beta} (\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) (Y G) I$
 - $=_{\beta} \text{if } I = 0 \text{ then } 1 \text{ else } I * (Y G) 0$
 - $=_{\beta} \text{if } I = 0 \text{ then } 1 \text{ else } I * (G (Y G) 0)$
 - $=_{\beta} \text{if } I = 0 \text{ then } 1 \text{ else } I * (\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)) (Y G) 0$
 - $=_{\beta} \text{if } I = 0 \text{ then } 1 \text{ else } I * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * (Y G) 0)$
 - $=_{\beta} I * I = I$

In Other Words

- The **Y** combinator “unrolls” or “unfolds” its argument an infinite number of times
 - $Y\ G = G\ (Y\ G) = G\ (G\ (Y\ G) = G\ (G\ (G\ (Y\ G))) = \dots$
 - **G** needs to have a “base case” to ensure termination
-
- We can use this trick to encode arbitrary recursion
-
- Note: this only works because we can evaluate in any order

Encodings

- Encodings are fun
- They show language expressiveness
- In practice, we usually add constructs as primitives
 - Much more efficient
 - Much easier to perform program analysis on and avoid silly mistakes with
 - E.g., our encodings of **true** and **0** are exactly the same, but we may want to forbid mixing booleans and integers

Lazy vs. Eager Evaluation

- Our non-deterministic reduction rule is fine for theory, but awkward to implement
-
- Two deterministic strategies:
 - *Lazy*: Given $(\lambda x.e1)\ e2$, do not evaluate $e2$ if x does not “need” $e1$
 - Also called left-most, call-by-name, call-by-need, applicative, normal-order (with slightly different meanings)
 - *Eager*: Given $(\lambda x.e1)\ e2$, always evaluate $e2$ fully before applying the function
 - Also called call-by-value

Lazy Operational Semantics

$$\frac{}{(\lambda x.e1) \rightarrow^l (\lambda x.e1)}$$

$$\frac{e1 \rightarrow^l \lambda x.e \quad e[e2/x] \rightarrow^l e'}{e1\ e2 \rightarrow^l e'}$$

- The rules are deterministic
- The rules do not reduce under λ
- The rules are normalizing:
 - If a is closed and there is a normal form b such that $a \rightarrow^* b$, then $a \rightarrow^l d$ for some d

Eager Operational Semantics

$$\frac{}{(\lambda x.e1) \rightarrow^e (\lambda x.e1)}$$

$$\frac{e1 \rightarrow^e \lambda x.e \quad e2 \rightarrow^e e' \quad e[e'/x] \rightarrow^e e''}{e1\ e2 \rightarrow^e e''}$$

- This semantics is also deterministic and does not reduce under λ
- But it is not normalizing
 - Example: $\text{let } x = \Delta\ \Delta \text{ in } (\lambda y.y)$

Lazy vs. Eager in Practice

- Lazy evaluation (call by name, call by need)
 - Has some nice theoretical properties
 - Terminates more often
 - Lets you play some tricks with “infinite” objects
 - Main example: Haskell
 -
- Eager evaluation (call by value)
 - Is generally easier to implement efficiently
 - Blends more easily with side effects
 - Main examples: Most languages (C, Java, ML, etc.)

Referential Transparency

- There are no *side effects* in the lambda calculus
 - The same expression always evaluates to the same result, regardless of the context
 - E.g., in $(f\ x)\ (f\ x)$, both calls always yield the same result
 - In contrast, consider this C example, where **a** and **b** may differ
 - ... $a=f(x); b=f(x); \dots$
- This means we can reason just like we would in mathematics, in a “pure” way

Functional Programming

- The λ calculus is a prototypical functional programming language:
 - Lots of higher-order functions
 - No side-effects
- In practice, many functional programming languages are “impure” and permit side-effects
 - But you’re supposed to avoid using them

Functional Programming Today

- Two main camps:
 - Haskell – Pure, lazy functional language; no side effects
 - ML (SML/NJ, O’Caml) – Call-by-value, with side effects
- Still around: LISP, Scheme
 - Disadvantage/advantage: No static type systems

O’Caml Tutorial

- Read-eval-print loop
- Basic data types (int, string)
- Let and letrec
- Lists
- Partial application (“upward funargs”)
 - A partially-applied function is called a *closure*
- Data structures (tagged unions)