Lambda Calculus and Functional Programming

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
  - | λx.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 (λx.e1) e2
  - Bind x to e2
  - Evaluate e1
  - Return the result of the evaluation
- This is called “beta-reduction”
  - (λx.e1) e2 → β e1[e2/x]
  - (λx.e1) e2 is called a redex
  - We’ll usually omit the beta
Three Conveniences

• Syntactic sugar for local declarations
  - let x = e1 in e2 is short for (λx.e2) e1

• Scope of λ extends as far to the right as possible
  - λx.λy.x y is λx.(λy.(x y))

• Function application is left-associative
  - x y z is (x y) z

Scoping and Parameter Passing

• Beta-reduction is not yet precise
  - (λx.e1) e2 → e1[e2/x]

• Example:
  - let x = a in let y = λz.x in let w = b in y x
  - which x's are bound to a, and which to b?

Free Variables and Alpha Conversion

• The set of free variables of a term is
  - FV(x) = {x}
  - FV(λx.e) = FV(e) - {x}
  - FV(e1 e2) = FV(e1) ∪ FV(e2)

• A term e is closed if FV(e) = Ø

• A variable that is not free is bound

Alpha Conversion

• Terms are equivalent up to renaming of bound variables
  - λx.e = λy.(e[y/x]) if y ≠ 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 ≠ x
  - (e1 e2)[e\x] = (e1[e\x] e2[e\x])
  - (λz.e1)[e\x] = λz.(e1[e\x]) if z ≠ x and z ≠ FV(e)

• Example:
  - (λx.y x) x =α (λw.y w) x

• (We won’t write alpha conversion down in the future)
**A Note on Substitutions**

- People write substitution many different ways
  - \( e_1[e_2 x] \)
  - \( e_1[x \mapsto e_2] \)
  - \( [x/e_2]e_1 \)
  - and more...
  - But they all mean the same thing

**Booleans**

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

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

**Combinators**

- Any closed term is also called a combinator
  - So \( \text{true} \) and \( \text{false} \) are both combinators

- Other popular combinators
  - \( I = \lambda x. x \)
  - \( S = \lambda x. \lambda y. x \ y \)
  - \( K = \lambda x. \lambda y. \lambda z. x \ z \ y \)

- Can also define calculi in terms of combinators
  - E.g., the SKI calculus
  - Turns out the SKI calculus is also Turing complete

**Pairs**

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

**Then**
- \( \text{fst} (a, b) \rightarrow^* a \)
- \( \text{snd} (a, b) \rightarrow^* b \)

**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

**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., \( n = \lambda x. \lambda y. <\text{apply } x \text{ times to } y> \)

- \( \text{succ} = \lambda z. \lambda x. \lambda y. (z \ x \ y) \)
- \( \text{iszero} = \lambda z. \ z \ (\lambda y. \text{false}) \text{ true} \)
**Natural Numbers (Scott)**

- \( 0 = \lambda x.\lambda y.x \)
- \( 1 = \lambda x.\lambda y.y\ 0 \)
- \( 2 = \lambda x.\lambda y.y\ 1 \)
- 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 Nonderministic Semantics**

- \( \text{(\lambda x.e1)\ e2} \rightarrow e1[e2]\x) \)
- \( \text{e} \rightarrow \text{e'} \)
- \( \text{(\lambda x.e) \rightarrow (\lambda x.e')} \)

**Natural Deduction**

- These are *natural deduction* style rules

**Example**

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

**The Church-Rosser Theorem**

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

Not Every Term Has a Normal Form

- Consider
  - $\Delta = \lambda x.x x$
  - Then $\Delta \Delta \rightarrow \Delta \Delta \rightarrow^*$
  - 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

- Fact $n = if n = 0 then 1 else n * fact(n-1)$
- Let $G = \lambda f.<body of factorial>$
  - I.e., $G = \lambda f.\lambda n.if n = 0 then 1 else n*f(n-1)$
  - $Y G I = \beta G (YG) I$
    - $=_\beta (\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) F$
    - $\rightarrow (\lambda x.F (x x)) (\lambda x.F (x x))$
    - $\rightarrow F ((\lambda x.F (x x)) (\lambda x.F (x x)))$
    - $\leftarrow F (Y F)$
In Other Words

• The Y combinator “unrolls” or “unfolds” its argument an infinite number of times
  ▪ \( Y \) \( G = G (Y G) = G (G (Y G)) = ... \)
  ▪ \( G \) 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

Lazy vs. Eager Evaluation

• Our non-deterministic reduction rule is fine for theory, but awkward to implement
  ▪ Two deterministic strategies:
    ▪ Lazy: Given \( (\lambda x. e_1) e_2 \), do not evaluate \( e_2 \) if \( x \) does not “need” \( e_1 \)
      - Also called left-most, call-by-name, call-by-need, applicative, normal-order (with slightly different meanings)
    ▪ Eager: Given \( (\lambda x. e_1) e_2 \), always evaluate \( e_2 \) fully before applying the function
      - Also called call-by-value

Lazy Operational Semantics

\[
\begin{align*}
(\lambda x. e_1) & \rightarrow^* (\lambda x. e_1) \\
 e_1 & \rightarrow^* \lambda x. e \quad e_2 \rightarrow^* e' \quad e'[x] \rightarrow^* e'' \\
 e_1 & \rightarrow^* e'
\end{align*}
\]

• The rules are deterministic
• The rules do not reduce under \( \lambda \)
• The rules are normalizing:
  ▪ If \( a \) is closed and there is a normal form \( b \) such that \( a \rightarrow^* b \), then \( a \rightarrow^* d \) for some \( d \)

Eager Operational Semantics

\[
\begin{align*}
(\lambda x. e_1) & \rightarrow^* (\lambda x. e_1) \\
 e_1 & \rightarrow^* \lambda x. e \quad e_2 \rightarrow^* e' \quad e'[x] \rightarrow^* e'' \\
 e_1 & \rightarrow^* e'
\end{align*}
\]

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

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 \( \text{true} \) and \( 0 \) are exactly the same, but we may want to forbid mixing booleans and integers

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:
      - \(a = f(x); \ b = f(x);\ ...\)
- This means we can reason just like we would in mathematics, in a "pure" way

Functional Programming

- The \(\lambda\) 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)