CMSC 631 – Program Analysis and Understanding
Spring 2006

Lambda Calculus

Motivation

- Commonly-used programming languages are large are 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 is Practical!

- An 8-bit microcontroller (Zilog Z8 encore board w/4KB SRAM) computing 1 + 1 using Church numerals in the Lambda calculus

```
LambdaCan version 1.1, Copyright (c) 2005 Timothy Jon Fraser
LambdaCan comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions; see the LICENSE file for details.
Enter formula:
((
(\M.(\N.(\s.(\z.((M s) ((N s) z))))))
(\s.(\z.(s z))))
(\s.(\z.(s z))))
;;
-> ((\N.(\s.(\z.(((\s.(\z.(sz)))s)((Ns)z)))))(\s.(\z.(sz))))
-> (\s.(\z.(((\s.(\z.(sz)))s)(((\s.(\z.(sz)))s)z))))
-> (\s.(\z.(((\s.(\z.(sz)))s)((\z.(sz))z))))
-> (\s.(\z.(((\s.(\z.(sz)))s)(sz))))
-> (\s.(\z.((\z.(sz))(sz))))
-> (\s.(\z.(s(sz))))
/>
Nodes used (out of 255):
        00040
Variables used:
        MNsz

Enter formula:
```

Tim Fraser

Origins of Lambda Calculus

- Invented in 1936 by Alonzo Church (1903-1995)
  - Princeton Mathematician
  - Lectures of lambda calculus published in 1941
  - Also know for
    - Church's Thesis
      - All effective computation is expressed by recursive (decidable) functions, i.e., in the lambda calculus
    - Church's Theorem
      - First order logic is undecidable

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 $\rightarrow_\beta$ 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]
  - what if there are multiple x's?

- Example:
  - let x = a in let y = λz.x in let x = b 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
    - let x = a in let y = λz.x in let x = b in y x
  - is "the same" as
    - let x = a in let y = λz.x in let w = b in y w
  - Variable names don't matter

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) = ∅

- 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:
  - $x[e\backslash x] = e$
  - $z[e\backslash x] = z$ if $z \neq x$
  - $(e1\ e2)[e\backslash x] = (e1[e\backslash x]\ e2[e\backslash x])$
  - $(\lambda z.e1)[e\backslash x] = \lambda z.(e1[e\backslash x])$ 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
  - $e1[e2\backslash x]$
  - $e1[x \mapsto e2]$
  - $[x/e2]e1$
  - 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
  - $(\lambda x.\lambda y.e)\ a\ b \rightarrow (\lambda y.e[a\backslash x])\ b \rightarrow e[a\backslash x][b\backslash y]$
- This is often called *Currying* and can be used to represent functions with any # of arguments

Booleans

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

Natural Numbers (Church)

- 0 = λx.λy.y
- 1 = λx.λy.x y
- 2 = λx.λy.x(x y)
- i.e., n = λx.λy.<apply x n times to y>

- succ = λz.λx.λy.x(z x y)
- iszero = λz.z (λy.false) true

Natural Numbers (Scott)

- 0 = λx.λy.x
- 1 = λx.λy.y 0
- 2 = λx.λy.y 1
- I.e., n = λx.λy.y (n-1)

- succ = λz.λx.λy.y z
- pred = λz.z 0 (λx.x)
- iszero = λz.z true (λx.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
  - (λx.e1) e2 → e1[e2\x]
  - Where does this rule apply?
    - Current answer: Anywhere within a term

A Nonderministic Semantics

$$\frac{}{(\lambda x.e1)\ e2 \to e1[e2\backslash x]} \qquad \frac{e \to e'}{(\lambda x.e) \to (\lambda x.e')}$$

$$\frac{e1 \to e1'}{e1\ e2 \to e1'\ e2} \qquad \frac{e2 \to e2'}{e1\ e2 \to e1\ e2'}$$

- The rules are a *small-step semantics*
  - It takes many →'s before we reach a normal form

Natural Deduction

- These are *natural deduction* style rules

$$\frac{H1 \quad H2 \quad \ldots \quad Hn}{C}$$

  - Read: If hypotheses H1 through Hn hold, then conclusion C holds

- The rules are axioms that define something, in this case what → means

- We will use this style of rule extensively

Example

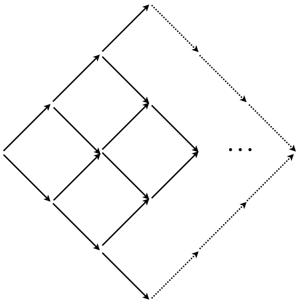
- We can apply reduction anywhere in a term
  - (λx.(λy.y) x ((λz.w) x) → λx.(x ((λz.w) x) → λx.x w
  - (λx.(λy.y) x ((λz.w) x) → λx.(λy.y x (w)) → λx.x w

- Does the order of evaluation matter?

The Church-Rosser Theorem

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

Proof



Normal Form

- A term is in *normal form* if it cannot be reduced
  - Examples: λx.x, λx.λy.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 $\to$
  - E.g., (λx.x) y → y ← (λz.λw.z) y y, so all three are beta equivalent
- If $a =_\beta b$, then there exists c such that $a \to^* c$ and $b \to^* c$
  - 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
  - Δ = λx.x x
  - Then Δ Δ → Δ Δ →···
- In general, *self application* leads to loops
  - ...which is good if we want recursion

A Fixpoint Combinator

- Also called a paradoxical combinator
  - Y = λf.(λx.f (x x)) (λx.f (x x))
  - Note: There are many versions of this combinator
- Then Y F $=_\beta$ F (Y F)
  - Y F = (λf.(λx.f (x x)) (λx.f (x x))) F
  - → (λx.F (x x)) (λx.F (x x))
  - → F ((λx.F (x x)) (λx.F (x x)))
  - ← F (Y F)

Example

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

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))) = ...
    - 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 (λx.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 (λx.e1) e2, always evaluate e2 fully before applying the function
        - Also called call-by-value

Lazy Operational Semantics

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

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

- The rules are deterministic and *big-step*
    - The right-hand side is reduced "all the way"
- 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

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

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

- This big-step semantics is also deterministic and does not reduce under λ
- But it is not normalizing
    - Example: let x = Δ Δ in (λy.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.)

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, OCaml) – Call-by-value, with side effects
- Still around: LISP, Scheme
  - Disadvantage/advantage: No static type systems

Call-by-Name Example

```
OCaml
let cond p x y = if p then x else y
let rec loop () = loop ()
let z = cond true 42 (loop ())
```

infinite loop at call

```
Haskell
cond p x y = if p then x else y
loop () = loop ()
z = cond True 42 (loop ())
```

3rd argument never used by cond, so never invoked

Two Cool Things to Do with CBN

- Build control structures with functions

```
cond p x y = if p then x else y
```

- “Infinite” data structures

```
integers n = n:(integers (n+1))
take 10 (integers 0)  (* infinite loop in cbv *)
```