CMSC 631 – Program Analysis and Understanding

Lambda Calculus
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
• 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
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:

\[ e ::= x \quad \text{variable} \]
\[ | \quad \lambda x.e \quad \text{function abstraction} \]
\[ | \quad e \ e \quad \text{function application} \]

• 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 $(\lambda x.e_1) \ e_2$
  - Bind $x$ to $e_2$
  - Evaluate $e_1$
  - Return the result of the evaluation

• This is called “beta-reduction”
  - $(\lambda x.e_1) \ e_2 \rightarrow_\beta e_1[e_2/x]$
  - $(\lambda x.e_1) \ e_2$ is called a redex
  - We’ll usually omit the beta
Three Conveniences

• Syntactic sugar for local declarations
  - \texttt{let x = e1 in e2} is short for \((\lambda x.e2) \ e1\)

• Scope of \(\lambda\) 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
  ▪ \((\lambda x.e_1) e_2 \rightarrow e_1[e_2/x]\)
  ▪ what if there are multiple x’s?

• Example:
  ▪ let x = a in
  ▪ let y = \lambda z.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 = \( \lambda z. x \) in let x = b in y x
  ▪ is “the same” as
    - let x = a in let y = \( \lambda z. x \) 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

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(\lambda x.e) &= FV(e) - \{x\} \\
FV(e_1 \ e_2) &= FV(e_1) \cup FV(e_2)
\end{align*}
\]

• 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 $y \notin \text{FV}(e)$

• 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 $z \neq x$
  - $(e_1 e_2)[e/x] = (e_1[e/x] e_2[e/x])$
  - $(\lambda z.e_1)[e/x] = \lambda z.(e_1[e/x])$ if $z \neq x$ and $z \notin \text{FV}(e)$

• 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
  ▪ \( e_1[e_2/x] \)
  ▪ \( e_1[x\mapsto e_2] \)
  ▪ \( [x/e_2]e_1 \)
  ▪ and more...

• But they all mean the same thing
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 → (\lambda x.\lambda y.x) \ b \ c → (\lambda y.b) \ c → b
  - if false then b else c → (\lambda x.\lambda y.y) \ b \ c → (\lambda y.y) \ c → 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., $n = \lambda x.\lambda y.\langle\text{apply } x \; n \; \text{times to } y\rangle$

• $\text{succ} = \lambda z.\lambda x.\lambda y.x(z \; x \; y)$
• $\text{iszero} = \lambda z.z \; (\lambda y.\text{false}) \; \text{true}$
Natural Numbers (Scott)

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

\[(\lambda x. e_1) \ e_2 \rightarrow e_1[e_2/x]\]

\[e \rightarrow e'\]

\[(\lambda x. e) \rightarrow (\lambda x. e')\]

\[e_1 \rightarrow e_1'\]

\[e_2 \rightarrow e_2'\]

\[e_1 \ e_2 \rightarrow e_1' \ e_2'\]

- Why are these semantics non-deterministic?
Example

- We can apply reduction anywhere in a term
  - \((\lambda x.(\lambda y.y) \times ((\lambda z.w) \times x)) \rightarrow \lambda x.(x \times ((\lambda z.w) \times x)) \rightarrow \lambda x.x \; w\)
  - \((\lambda x.(\lambda y.y) \times ((\lambda z.w) \times x)) \rightarrow \lambda x.(\lambda y.y \times (w)) \rightarrow \lambda x.x \; w\)

- Does the order of evaluation matter?
The Church-Rosser Theorem

• If \( a \rightarrow^* b \) and \( a \rightarrow^* c \), there exists \( d \) such that \( b \rightarrow^* d \) and \( c \rightarrow^* d \)

• Church-Rosser is also called confluence
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

• Notice that for our application rule, the argument need not be in normal form
Beta-Equivalence

• Let $\beta$ be the reflexive, symmetric, and transitive closure of $\to$
  • E.g., $(\lambda x.x) y \to y \leftarrow (\lambda z.\lambda w.z) y y$, so all three are beta equivalent

• If $a =_\beta b$, then there exists $c$ such that $a \to^* c$ 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
  ▪ $\Delta = \lambda x.x \ x$
  ▪ Then $\Delta \ \Delta \rightarrow \Delta \ \Delta \rightarrow \ldots$

• 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 = λf.<body of factorial>**
  - i.e., G = λf.λn.if n = 0 then 1 else n*f(n-1)
- **Y G 1 = \[G (YG) 1\]
  - = \[λf.λn.if n = 0 then 1 else n*f(n-1) (YG) 1\]
  - = \[if 1 = 0 then 1 else 1*((YG) 0)\]
  - = \[if 1 = 0 then 1 else 1*(G (YG) 0)\]
  - = \[if 1 = 0 then 1 else 1*(λf.λn.if n = 0 then 1 else n*f(n-1) (YG) 0)\]
  - = \[if 1 = 0 then 1 else 1*(if 0 = 0 then 1 else 0*((YG) 0))\]
  - = \[1*1 = 1\]
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.

But, only works because we’re call-by-name:

- Different combinator(s) for call-by-value
  - \( Z = \lambda f. (\lambda x. f \ (\lambda y. x \ x \ y)) \ (\lambda x. f \ (\lambda y. x \ x \ y)) \)
  - Why is this a fixed-point combinator? How does its difference from \( Y \) make it work for call-by-value?
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 \texttt{true} and \texttt{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 \((\lambda x. e_1) \ e_2\), 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

\[(\lambda x.e_1) \rightarrow^\prime (\lambda x.e_1)\]

\[e_1 \rightarrow^\prime \lambda x.e \quad e[e_2/x] \rightarrow^\prime e'\]

\[e_1 \quad e_2 \rightarrow^\prime e'\]

- The rules are deterministic and \textit{big-step}
  - The right-hand side is reduced “all the way”
- 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 $a \rightarrow^\prime b$
This big-step semantics is also deterministic and does not reduce under $\lambda$

- But it is not normalizing

  - Example: let $x = \Delta\Delta$ 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.)
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
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
- (RIP John McCarthy, died 24 Oct 2011, inventor of LISP.)
Influence of Functional Programming

- Functional ideas in many other languages
  - Garbage collection was first designed with Lisp; most languages often rely on a GC today
  - Generics in Java/C++ came from polymorphism in ML and from type classes in Haskell
  - Higher-order functions and closures (used widely in Ruby; proposed extension to Java) are pervasive in all functional languages
  - Many data abstraction principles of OO came from ML’s module system
  - ...

Wednesday, October 26, 2011
Call-by-Name Example

**OCaml**

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

**Haskell**

```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

infinite loop at call
Two Cool Things to Do with CBN

• Build control structures with functions

\[
\text{cond } p \ x \ y = \text{if } p \ \text{then } x \ \text{else } y
\]

• “Infinite” data structures

\[
\text{integers } n = n:\text{(integers } (n+1))
\]
\[
\text{take } 10 \ \text{integers } 0 \ \text{(* infinite loop in cbv *)}
\]