CS 461

Programming Language Concepts

Gang Tan
Computer Science and Engineering
Penn State University

Lambda Calculus

Readings

- Ch11.7 of the supplemental materials of the textbook
  - See the schedule page of the course website

History

- History
  - Introduced by Alonzo Church, Stephen Kleene
  - Greek letter lambda, which is used to introduce functions
  - No significance to the letter lambda
  - Calculus means there is a way to
    - calculate the result of applying functions to arguments
- Most PLs are rooted in lambda calculus
  - It provides a basic mechanism for function abstraction and application
  - Functional PLs: Lisp, ML, Haskell, other languages
  - Java, C++, and C# all support lambda functions
- Important part of CS history and foundations
- Warning:
  - We'll study formalism

Syntax

- t ::= x | λx. t | t1 t2
  - where x may be any variable
  - Function abstraction (function definition): λx. t
    - Define a new function whose parameter is x and whose body is t
    - Scheme: (lambda (x) t)
  - Function application (function call): t1 t2
    - t1 should eval to a function; t2 is the argument to the function
    - Scheme: (t1 t2)
    - Math: t1(t2)

Examples

- Function abstraction
  - λx. x
    - there is no need to write explicit returns; x is the returning result
  - λx. (x+3)
    - assume + is a built-in function
  - λf. λx. f (f x)
    - multi-parameter function, in curried notation
- Function application
  - (λx. x) 3 -> 3
  - (λx. (x+y)) 3 -> 3 + y
  - (λx. λy. (x+y)) 3 4 -> 3 + 4
  - (λz. (x + 2*y + z)) 5 -> x + 2*y + 5

Parsing convention

- The lambda-calculus grammar is ambiguous
  - E.g., t1 t2 t3 can be parsed in different ways
  - We'll use parentheses and associativity to disambiguate
- Convention
  - function abstraction: the scope of functions extends as far to the right as possible (unless encountering parentheses)
    - λf. f x = λf.(f x), not (λf. f) x
  - function application is left associative
    - f 2 3 = ((f 2) 3), not f (2 3), suppose f = λx. λy. x + y

Reduction (Informally)

- (λx. x) 3 = 3
  - using 3 to replace x
- (λy. (y+1)) 3
- (λf. λx. f (f x)) (λy. y+1)

= λx. (λy. y+1) ((λy. y+1) x)

= λx. (λy. y+1) (x+1)

= λx. (x+1)+1

- (λf. λx. f (f x)) (λy. y*y)
- (λx. x) (λz. z)
- (λx. x) (λx. x)

Review of Lam Calculus

- Theoretical foundation of FP: Lambda calculus
  - t ::= x | λx. t1 | t1 t2
  - λx. t is for function abstraction; x's scope is t
    - Functions in lambda calculus takes one parameter at a time
    - λf. λx. f (f x); in curried form
  - t1 t2 for function application
- Parsing convention
  - Function application left associative: t1 t2 t3 = (t1 t2) t3
  - Function's scope extends to the right as far as possible lambda x. f 3 = lambda x. (f 3)

9

Free and Bound Variables

- "λx. t" binds a new var x and its scope is t
  - Occurrences of x in t are said to be bound
    - Variable x is bound in λx. (x+y)
  - Bound variable is a "placeholder" and can be renamed
    - Function λx. (x+y) is the same function as λz. (z+y)
- Names of free (=unbound) variables matter
  - Variable y is free in λx. (x+y)
  - Function λx. (x+y) is *not* the same as λx. (x+z)
- Example: λx. ((λy. y+2) x) + y
  - y in "y+2" is bound, while the second occurrence of y is free

Formal def. of free variables

Goal: define FV(t), the set of free variables of t

FV(x) = {x}
$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$
FV(λx. t) = FV(t) – {x}

- FV(λx. x) = {}
- FV(λf. λx. f (g x)) = {g}
- Exercise
  - FV((λx. x) (λx. x))
  - FV(λx. ((λy. y+2) x) + y)

Alpha renaming (rename bound variables)

λx. t = λy. [y/x] t (α)
when y is not free in t

- λx. x = λy. y
- λx. ((λy. y+2) x) + y, rename the first y to z
  - Becomes λx. ((λz. z+2) x) + y
- λx. λy. x - y = λy. λx. y – x, rename x to y and y to x

Capture-Avoiding Substitution

Reduction (operational semantics):

(λx. t′) t → [t/x] t′

[t/x] x = t,
[t/x] y = y, where y is a variable different from x
[t/x] (t1 t2) = ([t/x] t1) ([t/x] t2)
[t/x] (λx. t1) = λx. t1
[t/x] (λy. t1) = λy. ([t/x] t1), where y is not free in t

- [3/y] (λx. x + y) = λx. x + 3
- [3/x] (λx. x + y) = λx. x + y
- [λx. x / x] x = λx. x

Rename Bound Variables

- Function application

  $\underbrace{(\lambda f.\ \lambda x.\ f\ (f\ x))}_{\text{apply twice}}\ \underbrace{(\lambda y.\ y+x)}_{\text{add x to argument}}$

- Substitute "blindly" and wrong result

  [(λy. y+x) / f] (λx. f (f x))

  = λx. [(λy. y+x) ((λy. y+x) x)] = λx. x+x+x

  Wrong step

- Rename bound variables

  (λf. λz. f (f z)) (λy. y+x)

  = λz. [(λy. y+x) ((λy. y+x) z))] = λz. z+x+x

Easy rule: always rename variables to be distinct

Reduction (Formal Semantics)

- Basic computation rule is β-reduction

  (λx. t′) t = [t/x] t′

  where substitution involves renaming as needed
- Reduction:
  - Apply the β-reduction rule to any subexpression
  - Repeat until no β-reduction is possible
- Normal form: a lambda-calculus term that cannot be further reduced

Reduction Maybe Nonderministic

- An example of two beta-reduction sequences
  - (λy. y+1) ((λy. y+1) 2) = (λy. y+1) (2 + 1)

    = (λy. y+1) 3 = 3+1 = 4
  - (λy. y+1) ((λy. y+1) 2) = ((λy. y+1) 2) + 1

    = (2+1) + 1 = 3+1 = 4
- Confluence (Church-Rosser theorem):
  - Final result (if there is one) is uniquely determined

β-Reduction Example

Ω Combinator: $\lambda x. (x\ x)$

Evaluate: Ω $(\lambda v. v)$ = $(\lambda x. (x\ x))\ (\lambda v. v)$
= $(\lambda v. v)\ (\lambda v. v)$ = $(\lambda v. v)$

Evaluate: Ω Ω = $(\lambda x. (x\ x))\ (\lambda x. (x\ x))$
= $(\lambda x. (x\ x))\ (\lambda x. (x\ x))$ = …

Infinite loop!

17

Programming in Lambda Calculus

18

Declarations as "Syntactic Sugar"

- Informal Examples
  - let x = 3 in x + 4
  - let x = 3 let y = 4 in x + y + y
  - let f = λ x. x+1 in f(3)
  - let g = λ f. λ x. f(f (x)) in
    let h = λ x. x+1
    g h 2
- Encoding of let
  - let x = N in M same as (λx. M) N
- Syntactic sugar: the let is sweeter to write, but we can think of it as a syntactic magic

Declarations as "Syntactic Sugar"

```
function f(x)
    return x+2
end;
f(5);
```

- same as let f =λx. x+2 in (f 5)

$\underbrace{(\lambda f.\ f(5))}_{\text{block body}}\ \underbrace{(\lambda x.\ x+2)}_{\text{declared function}}$

Extra reading: Tennent, *Language Design Methods Based on Semantics Principles.* Acta Informatica, 8:97-112, 197

Encoding: Boolean

Booleans

$\text{TRUE} \triangleq \lambda x.\lambda y.x \qquad \text{FALSE} \triangleq \lambda x.\lambda y.y$

Encoding "if" so that

$$\text{Spec: IF } b\ t1\ t2 = \begin{cases} t1 \text{ when } b \text{ is TRUE} \\ t2 \text{ when } b \text{ is FALSE} \end{cases}$$

$$\text{Definition: IF} \triangleq \lambda b.\lambda t1.\lambda t2.(b\ t1\ t2)$$

Check IF TRUE t1 t2 = t1 and IF FALSE t1 t2 = t2

Encoding: Boolean

Booleans

$\text{TRUE} \triangleq \lambda x.\lambda y.x \qquad \text{FALSE} \triangleq \lambda x.\lambda y.y$

Encoding of "and"

$$\text{Spec: AND } b_1\ b_2 = \begin{cases} \text{TRUE when } b_1, b_2 \text{ are both TRUE} \\ \text{FALSE otherwise} \end{cases}$$

$$\text{Definition: AND} \triangleq \lambda b_1.\lambda b_2.\ (b_1\ (b_2\ \text{TRUE FALSE})\ \text{FALSE})$$

Check AND TRUE TRUE = TRUE and
AND FALSE TRUE = FALSE

Encoding: Boolean

Booleans

$\text{TRUE} \triangleq \lambda x.\lambda y.x \qquad \text{FALSE} \triangleq \lambda x.\lambda y.y$

Encoding of "or"

$$\text{Spec: OR } b_1\ b_2 = \begin{cases} \text{TRUE when } \textit{either } b_1 \textit{ or } b_2 \text{ is TRUE} \\ \text{FALSE otherwise} \end{cases}$$

$$\text{Definition: OR} \triangleq \lambda b_1.\lambda b_2.\ (b_1\ \text{True}\ (b_2\ \text{TRUE FALSE}))$$

Check OR TRUE TRUE = TRUE and
OR FALSE FALSE = FALSE

Church Encoding of Numbers

Natural numbers

Church numerals: $\text{n} \triangleq \lambda f.\lambda z.\underbrace{f\ (f\ \ldots (f}_{\text{n invocations of f}}\ z)\ldots)$

$$\begin{aligned} 0 &\triangleq \lambda f.\lambda z.z \\ 1 &\triangleq \lambda f.\lambda z.(f\ z) \\ 2 &\triangleq \lambda f.\lambda z.(f\ (f\ z)) \\ &\ldots \end{aligned}$$

Church Numerals

Encoding of "+1":

SUCC $\triangleq$ $\lambda n. \lambda f. \lambda z. \; (f \; (n \; f \; z))$

Check "SUCC 2"= 3

Encoding of PLUS

PLUS $\triangleq$ $\lambda n_1 . \lambda n_2 . \; (n_1 \text{ SUCC } n_2)$

Check "PLUS 1 2" = 3

Multiplication and exponentiation can also be encoded.

Pure vs. Applied λ-Calculus

◆Pure λ-Calculus: the calculus discussed so far

◆Applied λ-Calculus:

- Built-in values and data structures
  - (e.g., 1, 2, 3, true, false, (1 2 3))
- Built-in functions
  - (e.g., +, ∗, /, and, or)
- Named functions
- Recursion