CS 242

Lambda Calculus

John Mitchell

Reading: Chapter 4

Announcements

- Homework 1
  - Will be posted on web today
  - Due Wed by 5PM, (*tentative*: file cabinet on first floor of Gates)
  - Additional "Historical Lisp" addendum on web in Handouts
- Discussion section
  - Friday 2:15-3:05 in Gates B03
- Office hours Monday and Tuesday (see web)
- Homework graders
  - We're looking for 6-10 students from this class
  - Send email to cs242@cs if interested
  - Thursday 5:30PM grading sessions

Theoretical Foundations

- Many foundational systems
  - Computability Theory
  - Program Logics
  - Lambda Calculus
  - Denotational Semantics
  - Operational Semantics
  - Type Theory
- Consider some of these methods
  - Computability theory (halting problem)
  - Lambda calculus (syntax, operational semantics)
  - Denotational semantics (later, in connection with tools)

Lambda Calculus

- Formal system with three parts
  - Notation for function expressions
  - Proof system for equations
  - Calculation rules called *reduction*
- Additional topics in lambda calculus
  - Mathematical semantics (=model theory)
  - Type systems

We will look at syntax, equations and reduction

There is more detail in the book than we will cover in class

History

- Original intention
  - Formal theory of substitution (for FOL, etc.)
- More successful for computable functions
  - Substitution --> symbolic computation
  - Church/Turing thesis
- Influenced design of Lisp, ML, other languages
  - See Boost Lambda Library for C++ function objects
- Important part of CS history and foundations

Why study this now?

- Basic syntactic notions
  - Free and bound variables
  - Functions
  - Declarations
- Calculation rule
  - Symbolic evaluation useful for discussing programs
  - Used in optimization (in-lining), macro expansion
    - Correct macro processing requires variable renaming
  - Illustrates some ideas about scope of binding
    - Lisp originally departed from standard lambda calculus, returned to the fold through Scheme, Common Lisp

Expressions and Functions

- Expressions
  x + y x + 2*y + z
- Functions
  λx. (x+y) λz. (x + 2*y + z)
- Application
  (λx. (x+y)) 3 = 3 + y
  (λz. (x + 2*y + z)) 5 = x + 2*y + 5

Parsing: λx. f (f x) = λx.(f (f (x)))

Higher-Order Functions

- Given function f, return function f ◦ f
  λf. λx. f (f x)
- How does this work?

(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

Same result if step 2 is altered.

Same procedure, Lisp syntax

- Given function f, return function f ◦ f
  (lambda (f) (lambda (x) (f (f x))))
- How does this work?
  ((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ y 1))

  = (lambda (x) ((lambda (y) (+ y 1))
  ((lambda (y) (+ y 1)) x))))

  = (lambda (x) ((lambda (y) (+ y 1)) (+ x 1))))

  = (lambda (x) (+ (+ x 1) 1))

JavaScript next slide

Same procedure, JavaScript syntax

- Given function f, return function f ◦ f
  function (f) { return function (x) { return f(f(x)); } ; }
- How does this work?

(function (f) { return function (x) { return f(f(x)); } ; }
(function (y) { return y +1; }))

function (x) { return (function (y) { return y +1; })
((function (y) { return y + 1; })
(x)); }

function (x) { return (function (y) { return y +1; }) (x + 1); }

function (x) { return ((x + 1) + 1); }

Declarations as "Syntactic Sugar"

```
function f(x) {
    return x+2;
}
f(5);
```

(λf. f(5)) (λx. x+2)

block body declared function

let x = e_1 in e_2 = (λx. e_2) e_1

Recall JavaScript

```
var u = { a:1, b:2 }
var v = { a:3, b:4 }
(function (x,y) {
    var tempA = x.a;
    var tempB =x.b;
    x.a=y.a; x.b=y.b;
    y.a=tempA; y.b-tempB
}) (u,v)
```

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

Free and Bound Variables

- Bound variable is "placeholder"
  – Variable x is bound in λx. (x+y)
  – Function λx. (x+y) is same function as λz. (z+y)
- Compare
  ∫ x+y dx = ∫ z+y dz ∀x P(x) = ∀z P(z)
- Name of free (=unbound) variable does matter
  – Variable y is free in λx. (x+y)
  – Function λx. (x+y) is *not* same as λx. (x+z)
- Occurrences
  – y is free and bound in λx. ((λy. y+2) x) + y

Reduction

- Basic computation rule is β-reduction

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

  where substitution involves renaming as needed

  (next slide)

- Reduction:
  - Apply basic computation rule to any subexpression
  - Repeat
- Confluence:
  - Final result (if there is one) is uniquely determined

Rename Bound Variables

- Function application

  $(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+x)$

  apply twice add x to argument

- Substitute "blindly"

  $\lambda x.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ x)]\ =\ \lambda x.\ x+x+x$

- Rename bound variables

  $(\lambda f.\ \lambda z.\ f\ (f\ z))\ (\lambda y.\ y+x)$

  $=\ \lambda z.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ z)]\ =\ \lambda z\ z+x+x$

Easy rule: always rename variables to be distinct

1066 and all that

- *1066 And All That*, Sellar & Yeatman, 1930

  1066 is a lovely parody of English history books, "Comprising all the parts you can remember including one hundred and three good things, five bad kings and two genuine dates."

- Battle of Hastings Oct. 14, 1066
  - Battle that ended in the defeat of Harold II of England by William, duke of Normandy, and established the Normans as the rulers of England

Main Points about Lambda Calculus

- λ captures "essence" of variable binding
  - Function parameters
  - Declarations
  - Bound variables can be renamed
- Succinct function expressions
- Simple symbolic evaluator via substitution
- Can be extended with
  - Types
  - Various functions
  - Stores and side-effects

  (But we didn't cover these)