CSE 505

Lecture #8

September 26, 2012

Data Representation

Booleans:

λx. λy. x	➔ true
λx. λy. y	➔ false

Natural numbers:

λ f . λ x . x	➔ 0
λ f . λ x . (f x)	➔ 1
λ f . λ x . (f (f x))	➔ 2
…	

let if = λb. λt. λe.((b t) e)

We can justify the above reptn by showing:

a. (((if true) T1) T2) ==>* T1

b. (((if false) T1) T2) ==>* T2

Example:

(((λb. λt. λe.((b t) e) true) T1) T2)
==> ((λt. λe.((true t) e) T1) T2)
==>* ((true T1) T2) = ((λx. λy.x T1) T2)
==> (λy.T1 T2)
==> T1

Idea behind Church Numerals

Constructors: zero, succ(zero), succ(succ(zero)), …

Alternatively: z, s(z), s(s(z)), …

Lisp Syntax: z, (s z), (s (s z)), …

Abstract Names: λs.λz.z,
λs.λz.(s z),
λs.λz.(s (s z)), …

Operations on numbers

Let succ = λn. λf. λx. ((n f) (f x))

Let add = λn1. λn2. λf. λx.
((n1 f) ((n2 f) x))

Let mult = λn1. λn2. λf. λx.
((n1 (n2 f)) x)

Let mystery = λn1. λn2. (n2 n1)

(succ λs.λz.z)

Data Structures

Recall Lisp lists:

‘(1) → (cons 1 nil)

‘(1 2 3) → (cons 1 (cons 2 (cons 3 nil)))

The names of the constructors nil and cons are not important, so we “abstract them away” in lambda calculus, as shown on next slide.

Encoding Lists

- λ c . λ n . n
- λ c . λ n . ((c tom) n)
- λ c. λ n . ((c tom) ((c dick) n))
- λ c. λ n . ((c tom) ((c dick) ((c harry) n)))
- ....

Function to get first element: λl.((l λx.λy.x) a)

(λl.((l λx.λy.x) a) λc.λn.((c tom) ((c dick) n)))

➔* tom

(λl.((l λx.λy.x) a) λc.λn.((c tom) ((c dick) n)))

⇒((λc.λn.((c tom) ((c dick) n))) λx.λy.x) a)

⇒(λn.((λx.λy.x tom) ((λx.λy.x dick) n))) a)

⇒(λn.(λy.tom ((λx.λy.x dick) n))) a)

⇒(λy.tom ((λx.λy.x dick) a)))

⇒tom

LAMBDA CALCULUS TOOL DEMO

Computability

- The language of lambda expressions is powerful enough to encode all computable functions!
- Notice that there is no recursive function definition – but this can be simulated, as will be next shown.

Recursive Definition

Consider recursive definition:

f(n) = if is0(n) then 1 else n * f(n-1)

Lisp syntax:

(defun f (n) (if (is0 n) 1 (* n (f (- n 1)))))

Lambda calculus (not quite):

letrec f = λn.(((if (is0 n)) 1)
((mult n) (f (pred n))))

Representing Recursion

let t = λf. λn.(((if (is0 n)) 1)
((mult n) (f (pred n))))

Fixed-Point Operator, Y:

let Y = λf. (λx.(f (x x)) (λx.(f (x x))

Note: Fixed point of F is an x such that F(x) = x

Thus the non-recursive equivalent of fact: (Y t)

Y = λf. (λx.(f (x x)) λx.(f (x x)))

Y is fixed-point operator, because (for any t):

(Y t) <==>* (t (Y t))

Derivation:

(Y t) = (λf. (λx.(f (x x)) λx.(f (x x))) t)

==> (λx.(t (x x)) λx.(t (x x)))

==> (t (λx.(t (x x)) λx.(t (x x)))))

<==> (t (Y t))

Recursion and Fixed-points

fact = λn.(((if (is0 n)) 1) ((mult n) (fact (pred n)))

t = λf. λn.(((if (is0 n)) 1) ((mult n) (f (pred n))))

Why does the fixed-point of t capture f?

Fixed point g has the property: g = (t g)

g = λn.(((if (is0 n)) 1) ((mult n) (g (pred n)))

Least Fixed Point

Consider: letrec f(n) = if n=0 then 0 else f(n);

Fixed-point $f1(n) = \left\{ \begin{array}{l} 0,\text{ if } n=0 \\ 1,\text{ if } n \geq 0 \end{array} \right\}$

Fixed-point $f2(n) = \left\{ \begin{array}{l} 0,\text{ if } n=0 \\ 2,\text{ if } n \geq 0 \end{array} \right\}$

Least fixed-point $g(n) = \left\{ \begin{array}{l} 0,\text{ if } n=0 \\ ?,\text{ if } n \geq 0 \end{array} \right\}$

Typed Lambda Calculi

Thus far, we have studied the untyped lambda calculus, i.e., no types associated with vars.

There are two well-known calculi:

- the simply-type lambda calculus
- the second-order (polymorphic) lambda calculus

Interesting, adding types causes all lambda expressions to terminate! Cannot have (x x).