CS 611: Lecture 8

Untyped Lambda Calculus
September 15, 1999

Cornell University Computer Science Department
Andrew Myers

Untyped Lambda Calculus

- IMP: no functions
- Lambda calculus: all functions

$e ::= x \mid e_0\ e_1 \mid \lambda x\, e_0$

x: *Identifier*. refers to variable defined by surrounding context.

$e_0\ e_1$: *Application*. Applies the function e_0 to the argument e_1

$\lambda x\ e_0$ *: Lambda expression*. Defines a new function with argument variable x and body e_0 (ala **fn** x => e_0)

- Universal, simple, core language (but *not Lisp/Scheme*)

Notation

- Lambda calculus is programming language *and* a mathematical notation.
- When programming language: fully parenthesized
- Identity function program: $(\lambda x\, x)$
- Mathematical identity function: $f(x) = x$ means $f = \lambda x : T.\ x$

Evaluation

- Application is evaluated by β-reduction:

$$((\lambda x_0\ e_1)\ e_2) \rightarrow e_1[e_2\ /\ x_0]$$

(defining how to substitute is tricky)

$$((\lambda x\, x)\ e) \rightarrow x\,[e\,/\,x\,] = e$$

$$(((\lambda x\,(\lambda y\ \ (y\,x\,)))\ 3)\ \mathrm{INC}) \rightarrow$$
$$((\lambda y\,(\,y\ 3))\ \mathrm{INC}) \rightarrow (\mathrm{INC}\ 3) \rightarrow 4$$

Operational semantics

- Large-step semantics, no store: configuration is an expression of the language

$$\frac{e_0 \rightarrow (\lambda\ x\ e_2)}{(e_0\ e_1) \rightarrow e_2[e_1/x]}$$

- Semantics using only β-reduction: any lambda term is a final result
- Call-by-name semantics: e_2 is not evaluated before substitution

Higher-order functions

- Can express functions that return (or accept) other functions easily
- A function that applies another function to 5: $(\lambda\ f\,(f\,5))$
- A function that returns a function that applies another function to its argument: $(\lambda\ v\,(\lambda\ f\,(f\,v)))$

Simulating multiple arguments

- Function in λ-calculus can be *curried* -- applied one argument at a time

$$(+\ 1\ 5) \rightarrow ((+\ 1)\ 5)$$

- Lambda calculus functions only need a single argument:

- $(\lambda\ (x_1 \ldots x_n)\ e) \rightarrow (\lambda\ x_1\ (\lambda \ldots\ (\lambda\ x_n\ e)\ldots))$
- $(e_0\ e_1\ e_2\ \ldots\ e_n) \rightarrow (\ldots((e_0\ e_1)\ e_2)\ldots\ e_n)$

Example

$(((\lambda\ x\ (\lambda\ y\ \ (y\ x)))\ 3)\ \mathrm{INC}) \rightarrow$
$((\lambda\ y\ (y\ 3))\ \mathrm{INC}) \rightarrow (\mathrm{INC}\ 3) \rightarrow 4$

Shorthand:

$((\lambda\ (x\ y)\ (y\ x))\ = (\lambda\ x\ (\lambda\ y\ \ (y\ x)))$

$((\lambda\ (x\ y)\ (y\ x))\ 3\ \mathrm{INC}) \rightarrow (\mathrm{INC}\ 3) \rightarrow 4$

Simulating let

- Lambda calculus has no "let" statement ala ML

(let $x = e_1$ in e_2) ⇨ $(\lambda\ x\ e_2)\ e_1$

Definitions

- Lambda calculus terms can become long; for compactness we will use names, multiple arguments as shorthand (not part of language!)

IDENTITY ≜ (λ x x)
INC ≜ (+ 1)
APPLY-TO-FIVE ≜ (λ f (f 5))
COMPOSE ≜ (λ (f g) (λ x (f (g x))))
TWICE ≜ (λ f (*COMPOSE* f f))
((*COMPOSE INC INC*) 2) → 4
((*TWICE* (*TWICE INC*)) 0) → 4

Representing booleans

- Lambda calculus is universal: no primitive boolean type is needed

TRUE ≜ (λ x (λ y x)) ≜ (λ (x y) x)
FALSE ≜ (λ x (λ y y)) ≜ (λ (x y) y)
if e_1 then e_2 else e_3 ≜ (*IF* e_1 e_2 e_3)
IF ≜ (λ (x y z) (x y z))
(*IF TRUE* e_2 e_3) → ((λ x (λ y x)) e_2) e_3)
→ ((λ y e_2) e_3) → e_2

- Call-by-name important! e_2 and e_3 are not evaluated eagerly by *IF*

Representing pairs (lists)

- Pair/list operations:

(*CONS* x y) : construct list with head x, tail y
(*FIRST* p) : return first item in list/first item in pair
(*REST* p) : return remainder of list/second item in pair

- One way to model these operations:

CONS ≜ (λ (x y) (λf (f x y)))
FIRST ≜ (λ p (p (λ(x y) x))) (= (λp (p *TRUE*)))
REST ≜ (λ p (p (λ(x y) y))) (= (λp (p *FALSE*)))

Natural numbers

- Model the number n as a function that composes an arbitrary function n times : *Church numerals*

$0 \triangleq (\lambda\,(f\,a)\;a)$ (= *FALSE*)

$1 \triangleq (\lambda\,(f\,a)\;(f\,a))$

$2 \triangleq (\lambda\,(f\,a)\;(f\,(f\,a)))$

$3 \triangleq (\lambda\,(f\,a)\;(f\,(f\,(f\,a))))$

$n \triangleq (\lambda\,(f\,a)\;\underbrace{(f\,(\ldots\,(f\,a)\ldots))}_{n \text{ times}})$

Arithmetic

- Can define *INC* function that adds one by writing a function that interposes an extra call to the function:

$n \triangleq (\lambda\,(f\,a)\;(f\,(\ldots\,(f\,a)\ldots)))$

$INC \triangleq (\lambda\,n\,(\lambda\,(f\,a)\;(f\,((n\,f)\,a))))$

Can define + and other arithmetic operators:

$+ \triangleq (\lambda\,(n_1\,n_2)\;(\lambda\,(f\,a)\;((n_1\,f)\;((n_2\,f)\,a)))$ *or*

$+ \triangleq (\lambda\,(n_1\,n_2)\;((n_1\,INC)\;n_2))$

Recursion

- How to express recursive functions?
- Consider factorial function:

$$\text{fact(x)} = \begin{cases} 1 & \text{if x = 0} \\ \text{x*fact(x-1)} & \text{if x > 0} \end{cases}$$

- Can't write this recursive definition!

FACT ≜
(λx (*IF* (*ZERO?* x) 1 (* x (*FACT* (- x 1)))))

Defining a recursive function

- Idea: introduce a function just like *FACT* except that it has an extra argument that should be passed a function f such that ((f f) x) computes factorial of x

FACT′ ≜
(λf (λx (*IF* (*ZERO?* x) 1 (* x ((f f) (- x 1))))))

- Now define *FACT* ≜ (*FACT′* FACT′)
- This expression diverges but its application to a Church numeral does not!

Evaluation of FACT

FACT ≜
(λf (λx (*IF* (*ZERO?* x) 1 (* x ((f f) (- x 1))))))
FACT ≜ (*FACT′* FACT′)

(*FACT* 2) → ((*FACT′* FACT′) 2)
→ (*IF* (*ZERO?* 2) 1 (* 2 ((*FACT′* FACT′) (- 2 1))))
→ (* 2 ((*FACT′* FACT′) (- 2 1))))))