CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 7: Lambda calculus
8 Sep 00

Administration

- HW1 due September 11 in class
  - modify *only* interpretation.sml
- New TA: James Cheney (jcheney@cs)

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
$\lambda\, x\, e_0$	**Abstraction/lambda term.** Defines a new function with argument variable x and body e_0 (ala ML's **fn** x => e_0)

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

Open vs. closed terms

- *term* = expression denoting a value
- Closed term: all identifiers bound by closest containing abstraction

(λ *x* … *x* (λ *y* … *y* …)…)

- Open term: some identifier(s) not bound: (λ x (y x))
- Legal lambda calculus programs: all closed terms

Evaluation

- Application is evaluated by β-reduction:

$$((\lambda\, x\, e_1)\; e_2) \rightarrow e_1\{e_2 / x\}$$

$e_1\,\{e_2/x\}$ means "e_1 with e_2 substituted for occurrences of x"
(**note:** defining "substituted" is tricky)

$((\lambda\, x\, x)\; e) \rightarrow x\,\{e / x\} = e$
$((\lambda\, x\, (\lambda\, x\, x))\; e) \rightarrow (\lambda\, x\, x)\,\{e / x\} = (\lambda\, x\, x)$
$(((\lambda\, x\, (\lambda\, y\, (y\, x)))\; 3)\; INC) \rightarrow$
$((\lambda\, y\, (y\, 3))\; INC) \rightarrow (INC\; 3) \rightarrow 4$

Higher-order functions

- Can express functions that return (or accept) other functions easily
  (all values *are only* functions)
- A function that applies another function to 5: (λ f (f 5))
- A function that returns a function that applies another function to its argument:
  (λ v (λ f (f v)))

Simulating multiple arguments

- Don't we need multiple arguments?

  $e ::= \ldots \mid e_0\ e_1 \ldots e_n \mid \lambda\ (x_1 \ldots x_n)\ e_0$
- Can *desugar* (rewrite syntactically) into single-argument calculus:

  $(\lambda\ (x_1 \ldots x_n)\ \ e\ ) \Rightarrow (\lambda\ x_1\ (\lambda \ldots (\lambda\ x_n\ e\ ) \ldots))$

  $(e_0\ e_1\ e_2\ \ldots\ e_n) \Rightarrow (\ldots((e_0\ e_1)\ e_2) \ldots\ e_n)$
- Multi-argument functions are *curried* – applied one argument at a time

  $(+\ 1\ 5) \Rightarrow ((+\ 1)\ 5)$

Example

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

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

Shorthand:

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

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

Operational semantics

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

$$\frac{e_0 \Downarrow \lambda\ x\ e_2 \qquad e_2\{e_1/x\} \Downarrow v}{e_0\ e_1 \Downarrow v}$$

- *Call-by-name* semantics: e_1 is not evaluated before substitution
- Call-by-name + β-reduction: any lambda term is a *value*: $v ::= \lambda\ x\ e$

Small-step semantics

$$\frac{}{(\lambda\ x\ e_1)\ e_2 \rightarrow e_1\{e_2\ /\ x\}}\ (\beta \text{ red'n})$$

$$\frac{e_1 \rightarrow e'_1}{e_1\ e_2 \rightarrow e'_1\ e_2}$$

call-by-name

$$\frac{e_1 \rightarrow e'_1}{e_1\ e_2 \rightarrow e'_1\ e_2} \qquad \frac{}{(\lambda\ x\ e_1)\ v \rightarrow e_1\{v\ /\ x\}}$$

$$\frac{e_2 \rightarrow e'_2}{v\ e_2 \rightarrow v\ e'_2}$$

call-by-value

An infinite loop

$(\lambda\ \text{x}\ (\text{x}\ \text{x}))\ (\lambda\ \text{x}\ (\text{x}\ \text{x})) \rightarrow ?$

This expression *diverges* (never stops taking small steps)

Simulating let

- Lambda calculus has no "let" statement ala ML

$(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) \Longrightarrow (\lambda\ x\ e_2)\ e_1$

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 or "if" statement 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 implement 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 ≜ (λ (*f a*) *a*) (= *FALSE*)

1 ≜ (λ (*f a*) (*f a*))

2 ≜ (λ (*f a*) (*f* (*f a*)))

3 ≜ (λ (*f a*) (*f* (*f* (*f a*))))

n ≜ (λ (*f a*) $\underbrace{(f\,(\ldots\,(f}_{n \text{ times}}\,a)\ldots)))$

Arithmetic

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

n ≜ (λ (*f a*) (f^n *a*))

INC ≜ (λ *n* (λ (*f a*) (*f* ((*n f*) *a*))))

Can define + and other arithmetic operators:

+ ≜ (λ (n_1 n_2) (λ (*f a*) ((n_1 *f*) ((n_2 *f*) *a*))) *or*

+ ≜ (λ (n_1 n_2) ((n_1 *INC*) n_2))