A λ-calculus with Constants and Let-blocks

Arvind
Laboratory for Computer Science
M.I.T.

September 16, 2002

http://www.csg.lcs.mit.edu/6.827

Interpreters

An *interpreter* for the λ-calculus is a program to reduce λ-expressions to "answers".

Two common strategies

- *applicative order:* left-most innermost redex
  aka call by value evaluation

- *normal order:* left-most (outermost) redex
  aka call by name evaluation

L4-3
Arvind

A Call-by-value Interpreter

Answers: WHNF
Strategy: leftmost-innermost redex but not inside a λ-abstraction

cv(E): Definition by cases on E

E = x | λx.E | E E

```
cv(x)       = x
cv(λx.E)    = λx.E
cv( E1 E2 ) = let f = cv(E1)
                  a = cv(E2)
              in
                  case f of
                      λx.E3 = cv(E3[a/x])
                      _     = (f a)
```

September 16, 2002 http://www.csg.lcs.mit.edu/6.827

L4-4
Arvind

A Call-by-name Interpreter

Answers: WHNF
Strategy: leftmost redex

cn(E): Definition by cases on E

E = x | λx.E | E E

```
cn(x)       = x
cn(λx.E)    = λx.E
cn(E1 E2)   = let f = cn(E1)
              in
                  case f of
                      λx.E3 = cn(E3[E2/x])
                      _     = (f E2)
```

September 16, 2002 http://www.csg.lcs.mit.edu/6.827

Normalizing Strategy

A *reduction strategy* is said to be *normalizing* if it terminates and produces an answer of an expression whenever the expression has an answer.

aka *the standard reduction*

Theorem: Normal order (left-most) reduction strategy is normalizing for the λ-calculus.

Example

(λx.y) ((λx.x x) (λx.x x))

call by value reduction

call by name reduction

For computing WHNF

the call-by-name interpreter is normalizing but the call-by-value interpreter is not

λ-calculus with Constants

$E ::= x \mid \lambda x.E \mid E\ E$
$\quad \mid Cond\ (E, E, E)$
$\quad \mid PF_k(E_1,...,E_k)$
$\quad \mid CN_0$
$\quad \mid CN_k(E_1,...,E_k)$

PF_1 ::= negate | not | ... | Prj_1| Prj_2 | ...
PF_2 ::= + | ...
CN_0 ::= Number | Boolean
CN_2 ::= cons | ...

It is possible to define *integers*, *booleans*, and *functions* on them in the pure λ-Calculus but the λ-calculus extended with constants is more useful as a programming language

Primitive Functions and Constructors

δ-*rules*

$+(\underline{n}, \underline{m}) \rightarrow \underline{n+m}$

...

Cond-rules

$Cond(True, e_1, e_2) \rightarrow e_1?$

$Cond(False, e_1, e_2) \rightarrow e_2$

Projection rules

$Prj_i(CN_k(e_1,...,e_k)) \rightarrow e_i$

λ-calculus with constants is confluent provided the new reduction rules are confluent

L4-9
Arvind

Constants and the η-rule

- η-rule no longer works for all expressions:

  $3 \neq \lambda x.(3\ \ x)$

  one cannot treat an integer as a function !

- η-rule is not useful if does not apply to all expressions because it is trivially true for λ-abstractions

  assuming $x \notin FV(\lambda y.M)$, is

  $\lambda x.(\lambda y.M\ x) = \lambda y.M$?

  $\lambda x.(\lambda y.M\ x)$

  $\rightarrow$

September 16, 2002 http://www.csg.lcs.mit.edu/6.827

L4-10
Arvind

Recursion

```
fact n = if (n == 0) then 1
                     else n * fact (n-1)
```

- fact can be rewritten as:

  fact = λn. Cond (Zero? n) 1 (Mul n (fact (Sub n 1)))

- *How to get rid of the* fact *on the RHS?*

  Idea: pass fact as an argument to itself

September 16, 2002 http://www.csg.lcs.mit.edu/6.827

Self-application and Paradoxes

Self application, i.e., (x x) is dangerous.

Suppose:

u ≡ λy. *if* (y y) = a *then* b *else* a

What is (u u) ?

Recursion and Fixed Point Equations

Recursive functions can be thought of as solutions of fixed point equations:

fact = λn. Cond (Zero? n) 1 (Mul n (fact (Sub n 1)))

Suppose

H = λf.λn.Cond (Zero? n) 1 (Mul n (f (Sub n 1)))

then

fact = H fact

fact is a *fixed point* of function H!

Fixed Point Equations

$f : D \rightarrow$?D

A fixed point equation has the form

$f(x) = x$

Its solutions are called the *fixed points* of f because if x_p is a solution then

$x_p = f(x_p) = f(f(x_p)) = f(f(f(x_p))) = \ldots$

Examples: $f: Int \rightarrow$?Int *Solutions*

$f(x) = x^2 - 2$

$f(x) = x^2 + x + 1$

$f(x) = x$

Least Fixed Point

Consider

f n = *if* n=0 *then* 1
else (*if* n=1 *then* f 3 *else* f (n-2))

$H = \lambda f.\lambda n.Cond(n=0 , 1, Cond(n=1, f\ 3, f\ (n-2))$

Is there an f_p such that $f_p = H\ f_p$?

Y : A Fixed Point Operator

$$Y \equiv \lambda f.(\lambda x.\ (f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))$$

Notice

$Y\ F \quad \rightarrow (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))$

$\rightarrow$

Mutual Recursion

```
odd  n = if n==0 then False else even (n-1)
even n = if n==0 then True  else odd  (n-1)
```

odd $= H_1$ even
even $= H_2$ odd
where
$H_1 = \lambda f.\lambda n.\text{Cond}(n=0, \text{False}, f(n-1))$
$H_2 = \lambda f.\lambda n.\text{Cond}(n=0, \text{True}, f(n-1))$

substituting "H_2 odd" for even
odd $= H_1\ (H_2$ odd$)$
$=$ H odd *where* H =
$\Rightarrow$ odd $=$ Y H

λ-calculus with Combinator Y

Recursive programs can be translated into the λ-calculus with constants and Y combinator. However,

- Y combinator violates every type discipline
- translation is messy in case of mutually recursive functions

  ⇒

  extend the λ-calculus with *recursive let blocks.*

λ_{let}: A λ-calculus with Letrec

Expressions

E ::= x | λx.E | E E | *let* S *in* E

Statements

S ::= ε | x = E | S; S

" ; " is associative and commutative

$S_1 ; S_2 \equiv S_2 ; S_1$

$S_1 ; (S_2 ; S_3) \equiv (S_1 ; S_2) ; S_3$

ε ; S ≡?S

let ε *in* E ≡ E

Variables on the LHS in a let expression must be pairwise distinct

α - Renaming

Needed to avoid the capture of free variables.

Assuming t is a new variable

$\lambda x.e \quad \equiv \lambda t.(e[t/x])$

let x = e ; S *in* e_0

$\equiv$ *let* t = e[t/x] ; S[t/x] *in* e_0[t/x]

where S[t/x] is defined as follows:

ε[t/x] = ε

(y = e)[t/x]= ?(y = e[t/x])

(S_1; S_2)[t/x]=? (S_1[t/x]; S_2[t/x])

(*let* S *in* e)[t/x]

=? (*let* S *in* e) *if* $x \notin$ FV(*let* S *in* e)

(*let* S[t/x] *in* e[t/x]) *if* $x \in$ FV(*let* S *in* e)

The β-rule

The normal β-rule

$(\lambda x.e)\ e_a \rightarrow$? e [e_a/x]

is replaced the following β-rule

$(\lambda x.e)\ e_a \rightarrow$?*let* t = e_a *in* e[t/x]

where t is a new variable

and *the Instantiation rules* which are used for substitution

λ_{let} Instantiation Rules

A free variable in an expression can be instantiated by a *simple expression*

V ::= λx.E *values*
SE ::= x | V *simple expression*

Instantiation rules

let x = a ; S *in* C[x] → *let* x = a ; S *in* C'[a]

simple expression

free occurrence of x in some context C

renamed C[ ] to avoid free-variable capture

(x = a ; SC[x]) → (x = a ; SC'[a])

x = a → x = C'[C[x]] *where* a = C[x]

Lifting Rules: Motivation

let
f = *let* S_1 *in* $\lambda x.e_1$
y = f a
in
((*let* S_2 *in* $\lambda x.e_2$) e_3)

How do we juxtapose

$(\lambda x.e_1)$ a

or

$(\lambda x.e_2)$ e_3 ?

Lifting Rules

In the following rules (*let* S’ *in* e’) is the α? *renaming* of (*let* S *in* e) to avoid name conflicts

x = *let* S *in* e → x = e’; S’

let S_1 *in* (*let* S *in* e) → *let* S_1; S’ *in* e’

(*let* S *in* e) e_1 → *let* S’ *in* e’ e_1

Cond((*let* S *in* e), e_1, e_2)
→ *let* S’ *in* Cond(e’, e_1, e_2)

PF_k(e_1,…,(*let* S *in* e),…,e_k)
→ *let* S’ *in* PF_k(e_1,…,e’,…,e_k)

Datastructure Rules

CN_k(e_1,…,e_k)
→ *let* t_1 = e_1; … ; t_k = e_k *in* <u>CN</u>$_k$(**t**$_1$,…,t_k)

Prj_i(<u>CN</u>$_k$(a_1,…,a_k))
→ a_i

Confluenence and Letrecs

odd = λn.Cond(n=0, False, even (n-1)) (M)
even = λn.Cond(n=0, True, odd (n-1))

substitute for even (n-1) in M
odd = λn.Cond(n=0, False,
Cond(n-1 = 0 , True, odd ((n-1)-1))) (M_1)
even = λn.Cond(n=0, True, odd (n-1))

substitute for odd (n-1) in M
odd = λn.Cond(n=0, False, even (n-1)) (M_2)
even = λn.Cond(n=0, True,
Cond(n-1 = 0 , False, even ((n-1)-1)))

M_1 and M_2 cannot be reduced to the same expression!

Proposition: λ_{let} is not confluent.

Ariola & Klop 1994

Contexts for Expressions

Expression Context for an expression

C[] ::= []
| λx.C[]
| C[] E | E C[]
| *let* S *in* C[]
| *let* SC[] *in* E

Statement Context for an expression

SC[] ::= x = C[]
| SC[] ; S | S; SC[]