A $\lambda$-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 $\lambda$-calculus is a program to reduce $\lambda$-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
A Call-by-value Interpreter

*Answers:* WHNF

*Strategy:* leftmost-innermost redex but not inside a \( \lambda \)-abstraction

\( cv(E) : \) Definition by cases on \( E \)

\[
E = x \mid \lambda x.E \mid E \ E
\]

- \( cv(x) = x \)
- \( cv(\lambda x.E) = \lambda x.E \)
- \( cv(E_1 E_2) = \text{let } f = cv(E_1) \)
  \[ a = cv(E_2) \]
  \[ \text{in} \]
  \[ \text{case } f \text{ of} \]
  \[ \lambda x.E_3 = cv(E_3[a/x]) \]
  \[ \_ = (f \ a) \]

A Call-by-name Interpreter

*Answers:* WHNF

*Strategy:* leftmost redex

\( cn(E) : \) Definition by cases on \( E \)

\[
E = x \mid \lambda x.E \mid E \ E
\]

- \( cn(x) = x \)
- \( cn(\lambda x.E) = \lambda x.E \)
- \( cn(E_1 E_2) = \text{let } f = cn(E_1) \)
  \[ \text{in} \]
  \[ \text{case } f \text{ of} \]
  \[ \lambda x.E_3 = cn(E_3[E_2/x]) \]
  \[ \_ = (f \ E_2) \]
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 $\lambda$-calculus.

Example

\[(\lambda x. y)((\lambda x. x) (\lambda x. x))\]

call by value reduction  \hspace{1cm} call by name reduction

For computing WHNF

the call-by-name interpreter is normalizing but the call-by-value interpreter is not
\[\lambda\text{-calculus with Constants}\]

\[
E ::= x \mid \lambda x. E \mid E \ E \ \\
\mid \text{Cond} (E, E, E) \ \\
\mid \text{PF}_k (E_1, \ldots, E_k) \ \\
\mid \text{CN}_0 \ \\
\mid \text{CN}_k (E_1, \ldots, E_k)
\]

\[
\text{PF}_1 ::= \text{negate} \mid \text{not} \mid \ldots \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \ldots
\]

\[
\text{PF}_2 ::= + \mid \ldots
\]

\[
\text{CN}_0 ::= \text{Number} \mid \text{Boolean}
\]

\[
\text{CN}_2 ::= \text{cons} \mid \ldots
\]

It is possible to define \textit{integers}, \textit{booleans}, and \textit{functions} on them in the pure \(\lambda\)-Calculus but the \(\lambda\)-calculus extended with constants is more useful as a programming language.

\[\text{Primitive Functions and Constructors}\]

\[
\text{\(\delta\)-rules} \ \\
+( n, m) \rightarrow n+m
\]

\[
\text{Cond-rules} \ \\
\text{Cond} (\text{True}, e_1, e_2) \rightarrow e_1? \ \\
\text{Cond} (\text{False}, e_1, e_2) \rightarrow e_2
\]

\[
\text{Projection rules} \ \\
\text{Prj}_i (\text{CN}_k (e_1, \ldots, e_k)) \rightarrow e_i
\]

\(\lambda\)-calculus with constants is confluent provided the new reduction rules are confluent.
Constants and the \( \eta \)-rule

- \( \eta \)-rule no longer works for all expressions:
  \[ 3 \neq \lambda x.(3 \ x) \]
  *one cannot treat an integer as a function!*

- \( \eta \)-rule is not useful if does not apply to all expressions because it is trivially true for \( \lambda \)-abstractions

\[
\text{assuming } x \notin \text{FV}(\lambda y.M), \text{ is} \\
\lambda x. (\lambda y.M \ x) = \lambda y.M \\
\lambda x. (\lambda y.M \ x) \\
\rightarrow
\]

Recursion

- fact can be rewritten as:
  \[
  \text{fact } n = \text{if } (n == 0) \text{ then } 1 \\
  \text{else } n \ast \text{fact } (n-1)
  \]

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

  Idea: pass fact as an argument to itself
Self-application and Paradoxes

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

Suppose:

\[ u \equiv \lambda y. \text{if } (y \ y) = a \text{ then } b \text{ else } a \]

What is \((u \ u)\)?

---

Recursion and Fixed Point Equations

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

\[ \text{fact} = \lambda n. \text{Cond} (\text{Zero? } n) \ 1 \ (\text{Mul} \ n \ (\text{fact} \ (\text{Sub} \ n \ 1))) \]

Suppose

\[ H = \lambda f. \lambda n. \text{Cond} (\text{Zero? } n) \ 1 \ (\text{Mul} \ n \ (f \ (\text{Sub} \ n \ 1))) \]

then

\[ \text{fact} = H \ \text{fact} \]

fact is a fixed point of function H!
Fixed Point Equations

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: \( f: \text{Int} \to \text{Int} \)

Solutions

- \( f(x) = x^2 - 2 \)
- \( f(x) = x^2 + x + 1 \)
- \( f(x) = x \)

Least Fixed Point

Consider
\[
f(n) = \begin{cases} 
1 & \text{if } n=0 \\
3 & \text{if } n=1 \\
f(n-2) & \text{else}
\end{cases}
\]
\[ H = \lambda f. \lambda n. \text{Cond}(n=0, 1, \text{Cond}(n=1, 3, f(n-2))) \]

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)

\[
\begin{align*}
\text{odd} & = H_1, \text{even} \\
\text{even} & = H_2, \text{odd}
\end{align*}
\]

where
\[
\begin{align*}
H_1 & = \lambda f.\lambda n.\text{Cond}(n=0, \text{False}, f(n-1)) \\
H_2 & = \lambda f.\lambda n.\text{Cond}(n=0, \text{True}, f(n-1))
\end{align*}
\]

substituting “H_2 odd” for even
\[
\begin{align*}
\text{odd} & = H_1 (H_2 \text{ odd}) \\
& = H \text{ odd} \quad \text{where} \quad H = \\
\Rightarrow \delta \text{dd} & = Y H
\end{align*}
\]
\(\lambda\)-calculus with Combinator \(Y\)

Recursive programs can be translated into the \(\lambda\)-calculus with constants and \(Y\) combinator. However,

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

\(\implies\) extend the \(\lambda\)-calculus with \textit{recursive let blocks}.

\(\lambda_{let}\): A \(\lambda\)-calculus with \textit{Letrec}

\[E ::= x \mid \lambda x. E \mid E \ E \mid \text{let } S \text{ in } E\]

\[S ::= \epsilon \mid x = E \mid S; S\]

\(\text{"};\text{" is associative and commutative}\)

\[S_1 ; S_2 \equiv S_2 ; S_1\]
\[S_1 ; (S_2 ; S_3) \equiv (S_1 ; S_2) ; S_3\]
\[\epsilon ; S \equiv \epsilon\]
\[\text{let } \epsilon \text{ in } E \equiv E\]

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

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

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

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

$$\varepsilon[t/x] = \varepsilon$$

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

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

$$(let \ S \ in \ e)[t/x] = \? \ (let \ S[t/x] \ in \ e[t/x])$$

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

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

The **β-rule**

The normal β-rule

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

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
\(\lambda_{let}\) Instantiation Rules

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

\[ V ::= \lambda x. E \quad \text{values} \]

\[ SE ::= x \mid V \quad \text{simple expression} \]

Instantiation rules

\[ \text{let } x = a \ ; S \ \text{in } C[x] \rightarrow \ \text{let } x = a \ ; S \ \text{in } C'[a] \]

- simple expression
- free occurrence of \(x\) in some context \(C\)
- renamed \(C[\ ]\) to avoid free-variable capture

\[ (x = a \ ; SC[x]) \rightarrow (x = a \ ; SC'[a]) \]

\[ x = a \rightarrow x = C'[C[x]] \quad \text{where } a = C[x] \]

Lifting Rules: Motivation

\[ \text{let } \]

\[ f = \text{let } S_1 \ \text{in } \lambda x. e_1 \]

\[ y = f \ a \]

\[ \text{in} \]

\[ ((\text{let } S_2 \ \text{in } \lambda x. e_2) \ e_3) \]

\[ \text{How do we juxtapose} \]

\[ (\lambda x. e_1) \ a \]

or

\[ (\lambda x. e_2) \ e_3 \quad ? \]
Lifting Rules

In the following rules \((\text{let } S' \text{ in } e')\) is the \(\alpha\)-renaming of \((\text{let } S \text{ in } e)\) to avoid name conflicts

\[
\begin{align*}
x &= \text{let } S \text{ in } e & \rightarrow & \ x = e' ; S' \\
\text{let } S_1 \text{ in } (\text{let } S \text{ in } e) & \rightarrow & \ \text{let } S_1 ; S' \text{ in } e' \\
(\text{let } S \text{ in } e) \ e_1 & \rightarrow & \ \text{let } S' \text{ in } e' \ e_1 \\
\text{Cond}(\text{let } S \text{ in } e), e_1, e_2) & \rightarrow & \ \text{let } S' \text{ in } \text{Cond}(e', e_1, e_2) \\
\text{PF}_k(e_1, \ldots, (\text{let } S \text{ in } e), \ldots, e_k) & \rightarrow & \ \text{let } S' \text{ in } \text{PF}_k(e_1, \ldots, e', \ldots, e_k)
\end{align*}
\]

Datastructure Rules

\[
\begin{align*}
\text{CN}_k(e_1, \ldots, e_k) & \rightarrow \ \text{let } t_1 = e_1 ; \ldots ; t_k = e_k \text{ in } \text{CN}_k(t_1, \ldots, t_k) \\
\text{Pr}_{ij}(\text{CN}_k(a_1, \ldots, a_k)) & \rightarrow \ a_i
\end{align*}
\]
Confluence and Letrecs

odd = \lambda n.\text{Cond}(n=0, \text{False}, \text{even }(n-1)) \quad (M)
even = \lambda n.\text{Cond}(n=0, \text{True}, \text{odd }(n-1))

*substitute for even *(n-1) in M*

odd = \lambda n.\text{Cond}(n=0, \text{False}, \text{Cond}(n-1 = 0 , \text{True}, \text{odd }((n-1)-1))) \quad (M_1)
even = \lambda n.\text{Cond}(n=0, \text{True}, \text{odd }(n-1))

*substitute for odd *(n-1) in M*

odd = \lambda n.\text{Cond}(n=0, \text{False}, \text{even }(n-1)) \quad (M_2)
even = \lambda n.\text{Cond}(n=0, \text{True}, \text{Cond}(n-1 = 0 , \text{False}, \text{even }((n-1)-1)))

M_1 and M_2 cannot be reduced to the same expression!

Proposition: \(\lambda\text{let}\) is not confluent.  

*Ariola & Klop 1994*

Contexts for Expressions

Expression Context for an expression

\[ C[] ::= [] \]
\[ \quad | \lambda x. C[] \]
\[ \quad | C[] E \quad | E C[] \]
\[ \quad | \text{let } S \text{ in } C[] \]
\[ \quad | \text{let } SC[] \text{ in } E \]

Statement Context for an expression

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