A $\lambda$-calculus with Constants and Let-blocks

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

September 19, 2006

Outline

- Recursion and Y combinator
- The $\lambda_{let}$ Calculus
Recursion

\[ \text{fact } n = \begin{cases} 1 & \text{if } (n == 0) \\ n \times \text{fact } (n-1) & \text{else} \end{cases} \]

- fact can be rewritten as:
  \[ \text{fact } = \lambda n. \text{Cond } \text{(Zero? } n) \ 1 \ (\text{Mul } n \ (\text{fact } (\text{Sub } n 1))) \]

- \textbf{How to get rid of the fact on the RHS?}

  Idea: pass fact as an argument to itself
  \[ H = \lambda f. \lambda n. \text{Cond } \text{(Zero? } n) \ 1 \ (\text{Mul } n \ (f \ f \ (\text{Sub } n 1))) \]
  \[ \text{fact } = H \ H \]

  \textit{Self application!}

Self-application and Paradoxes

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

Suppose:
\[ u = \lambda y. \begin{cases} b & \text{if } (y \ y) = a \\ a & \text{else} \end{cases} \]

What is \((u \ u)\) ?
\[ (u \ u) \to \begin{cases} b & \text{if } (u \ u) = a \\ a & \text{else} \end{cases} \]

\textit{Contradiction!!!}

Any semantics of \(\lambda\)-calculus has to make sure that functions such as \(u\) have the meaning \(\perp\), i.e.
“totally undefined” or “no information”.

Self application also violates every type discipline.
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

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

\[ \text{fact} \] 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

\[ x_p = f(x_p) = f(f(x_p)) = f(f(f(x_p))) = ... \]

Examples:

<table>
<thead>
<tr>
<th>f(x) = Int → Int</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^2 - 2 )</td>
<td>( x = 2, x = -1 )</td>
</tr>
<tr>
<td>( x^2 + x + 1 )</td>
<td>no solutions</td>
</tr>
<tr>
<td>( x )</td>
<td>infinite number of solutions</td>
</tr>
</tbody>
</table>
Least Fixed Point

Consider

\[ f \ n = \begin{cases} 1 & \text{if } n = 0 \\ \text{else} \ (f \ 3 \ \text{if } n = 1 \ \text{else} \ f \ n - 2) \end{cases} \]

\[ H = \lambda f. \lambda n. \text{Cond}(n = 0, 1, \text{Cond}(n = 1, f \ 3, f \ n - 2)) \]

Is there an \( f_p \) such that \( f_p = H \ f_p \)?

\[
\begin{array}{c|c|c}
 f1 \ n & \text{if } n \text{ is even} & \text{otherwise} \\
 \hline
 1 & \Downarrow & \perp \\
\end{array}
\]

\[
\begin{array}{c|c|c}
 f2 \ n & \text{if } n \text{ is even} & \text{otherwise} \\
 \hline
 1 & \Downarrow & 5 \\
\end{array}
\]

\( f1 \) contains no arbitrary information and is called the least fixed point. *Unique solution!*

Y : A Fixed Point Operator

\[ Y \equiv \lambda f. (\lambda x. (f \ (x \ x))) \ (\lambda x. (f \ (x \ x))) \]

Notice

\[ Y \ F \rightarrow \lambda x. (F \ (x \ x)) \ (\lambda x. (F \ (x \ x))) \]

\[ \rightarrow F \ (\lambda x. (F \ (x \ x))) \ (\lambda x. (F \ (x \ x))) \]

\[ F \ (Y \ F) \rightarrow F \ (\lambda x. (F \ (x \ x))) \ (\lambda x. (F \ (x \ x))) \]

\[ F \ (Y \ F) = Y \ F \quad (Y \ F) \text{ is a fixed point of } F \]

*Y computes the least fixed point of any function!*  
*There are many different fixed point operators.*
Mutual Recursion

odd \ n = \text{ if } \ n == 0 \ \text{ then } \text{False} \ \text{ else } \text{even} \ (n-1) \\
even \ n = \text{ if } \ n == 0 \ \text{ then } \text{True} \ \text{ else } \text{odd} \ (n-1)

\begin{align*}
\text{odd} & = H_1 \ \text{even} \\
\text{even} & = H_2 \ \text{odd} \\
\text{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))
\end{align*}

Can we expressing odd using Y?

$$\Rightarrow \text{odd} = \lambda H. H_1 (H_2 f)$$

\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

$$\Rightarrow \text{extend the } \lambda\text{-calculus with recursive let blocks.}$$
Outline

- Recursion and Y combinator
- The \( \lambda \)let Calculus

\( \lambda \)-calculus with Constants & Letrec

\[
E ::= x \mid \lambda x.E \mid E \; E \\
| \text{Cond} (E, E, E) \\
| \text{PF}_k(E_1, \ldots, E_k) \\
| \text{CN}_0 \\
| \text{CN}_k(E_1, \ldots, E_k) \mid \text{CN}(SE_1, \ldots, SE_k) \\
| \text{let } S \text{ in } E
\]

\[
\text{PF}_1 ::= \text{negate} \mid \text{not} \mid \ldots \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \ldots \\
\text{PF}_2 ::= + \mid \ldots \\
\text{CN}_0 ::= \text{Number} \mid \text{Boolean} \\
\text{CN}_2 ::= \text{cons} \mid \ldots
\]

**Statements**
- \( S ::= \varepsilon \mid x = E \mid S; S \)

*Variables on the LHS in a let expression must be pairwise distinct*
Let-block Statements

"\;" is associative and commutative

\[ S_1 \; S_2 \equiv S_2 \; S_1 \]

\[ S_1 \; (S_2 \; S_3) \equiv (S_1 \; S_2) \; S_3 \]

\[ \varepsilon \; S \equiv S \]

\[ let \; \varepsilon \; in \; E \equiv E \]

Free Variables of an Expression

\[ FV(x) = \{x\} \]

\[ FV(E_1 E_2) = FV(E_1) \cup FV(E_2) \]

\[ FV(\lambda x.E) = FV(E) - \{x\} \]

\[ FV(let \; S \; in \; E) = FVS(S) \cup FV(E) - BVS(S) \]

\[ FVS(\varepsilon) = \{\} \]

\[ FVS(x = E; \; S) = FV(E) \cup FVS(S) \]

\[ BVS(\varepsilon) = \{\} \]

\[ BVS(x = E; \; S) = \{x\} \cup BVS(S) \]
\[ \alpha \text{-Renaming (to avoid free variable capture)} \]

Assuming \( t \) is a new variable, rename \( x \) to \( t \):

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

\[
\text{let } x = e \; ; \; S \text{ in } e_0
\]

\[
\equiv \text{let } t = e[t/x] \; ; \; S[t/x] \text{ in } e_0[t/x]
\]

where \( [t/x] \) is defined as follows:

\[
\begin{align*}
x[t/x] &= t \\
y[t/x] &= y \quad \text{if } x \neq y \\
(E_1 E_2)[t/x] &= (E_1[t/x] \ E_2[t/x]) \\
(\lambda x . E)[t/x] &= \lambda x . E \\
(\lambda y . E)[t/x] &= \lambda y . E[t/x] \quad \text{if } x \neq y \\
(\text{let } S \text{ in } E)[t/x] &= (\text{let } S \text{ in } E) \quad \text{if } x \notin \text{FV}(\text{let } S \text{ in } E) \\
&= (\text{let } S[t/x] \text{ in } E[t/x]) \quad \text{if } x \in \text{FV}(\text{let } S \text{ in } E)
\end{align*}
\]

---

**Primitive Functions and Datastructures**

\[ \delta \text{-rules} \]

\[
+ (n, m) \rightarrow n + m
\]

\[ \text{Cond-rules} \]

\[
\begin{align*}
\text{Cond(True, } e_1, e_2 \text{ )} & \rightarrow e_1 \\
\text{Cond(False, } e_1, e_2 \text{ )} & \rightarrow e_2
\end{align*}
\]

\[ \text{Data-structures} \]

\[
\begin{align*}
\text{CN}_k(e_1, \ldots, e_k) & \rightarrow \left\{ \begin{array}{ll}
\text{let } t_1 = e_1; \ldots ; t_k = e_k \\
\text{in } \text{CN}_k(t_1, \ldots, t_k)
\end{array} \right. \\
\text{Prj}_i(\text{CN}_k(a_1, \ldots, a_k)) & \rightarrow a_i
\end{align*}
\]
The $\beta$-rule

The normal $\beta$-rule

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

is replaced by the following $\beta$-rule

$$(\lambda x. e) e_a \rightarrow \text{let } t = e_a \text{ in } e[t/x]$$

where $t$ is a new variable

and the Instantiation rules which are used to refer to the value of a variable

---

Values and Simple Expressions

**Values**

$$V ::= \lambda x. E \mid \text{CN}_0 \mid \text{CN}_k(SE_1, \ldots, SE_k)$$

**Simple expressions**

$$SE ::= x \mid V$$
Contexts for Expressions

A context is an expression (or statement) with a “hole” such that if an expression is plugged in the hole the context becomes a legitimate expression:

\[
C[] ::= [\]
| \lambda x.C[]
| C[] E | E C[]
| let S in C[]
| let SC[] in E
\]

Statement Context for an expression

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

\[\lambda\]_\text{let} Instantiation Rules

A free variable in an expression can be instantiated by a \textit{simple expression}

Instantiation rule 1
\[ (let \ x = a \ ; S \ in \ C[x]) \rightarrow (let \ x = a \ ; S \ in \ C'[a]) \]

\begin{itemize}
  \item \text{simple expression}
  \item \text{free occurrence of } x \text{ in some context } C
  \item \text{renamed } C[ ] \text{ to avoid free-variable capture}
\end{itemize}

Instantiation rule 2
\[ (x = a \ ; SC[x]) \rightarrow (x = a \ ; SC'[a]) \]

Instantiation rule 3
\[ x = a \quad \rightarrow \quad x = C'[C[x]] \]

\textit{where} \ a = C[x]
Lifting Rules: Motivation

\[
\begin{align*}
\text{let} & \\
f &= \text{let } S_1 \text{ in } \lambda x. e_1 \\
y &= f \ a \\
in & ((\text{let } S_2 \text{ in } \lambda x. e_2) \ e_3)
\end{align*}
\]

How do we juxtapose 
\[
(\lambda x. e_1) \ a
\]
or
\[
(\lambda x. e_2) \ e_3
\]

Lifting Rules

(\text{let } S' \text{ in } e') is the $\alpha$-renamed (let S in e) to avoid name conflicts in the following rules:

\[
\begin{align*}
x &= \text{let } S \text{ in } e & \rightarrow & x = e'; S'
\end{align*}
\]

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

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

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

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

Can odd in M_1 and M_2 be reduced to the same expression?

Proposition: $\lambda_{\text{let}}$ is not confluent.

Ariola & Klop 1994

\[ \lambda \text{ versus } \lambda_{\text{let}} \text{ Calculus} \]

Terms of the $\lambda_{\text{let}}$ calculus can be translated into terms of the $\lambda$ calculus by systematically eliminating the let blocks. Let T be such a translation.

Suppose $e \rightarrow e_1$ in $\lambda_{\text{let}}$ then does there exist a reduction such that $T[[e]] \rightarrow T[[e_1]]$ in $\lambda$?

We need a notion of observable values to compare terms in a meaningful way.
Instantaneous Information

“Instantaneous information” (info) of a term is defined as a (finite) trees

\[ T_p ::= \perp | \lambda | CN_0 | CN_k(T_{p1},...,T_{pk}) \]

Info: \( E \rightarrow T_p \)

\[
\begin{align*}
\text{Info}\{S \text{ in } E\} &= \text{Info}\ [E] \\
\text{Info}\[\lambda x. E\] &= \lambda \\
\text{Info}\[CN_0\] &= CN_0 \\
\text{Info}\[CN_k(a_1,...,a_k)\] &= CN_k(\text{Info}[a_1],...,\text{Info}[a_k]) \\
\text{Info}[E] &= \perp \quad \text{otherwise}
\end{align*}
\]

Notice this procedure always terminates

Reduction and Info

Terms can be compared by their Info value

\[
\begin{align*}
\perp & \leq t \quad \text{(bottom)} \\
t & \leq t \quad \text{(reflexive)} \\
CN_k(v_1,...,v_i,...,v_k) & \leq CN_k(v_1,...,v'_i,...,v_k) \quad \text{if } v_i \leq v'_i
\end{align*}
\]

**Proposition** Reduction is monotonic wrt Info:

If \( e \rightarrow e_1 \) then \( \text{Info}[e] \leq \text{Info}[e_1] \).

**Proposition** Confluence wrt Info:

If \( e \rightarrow e_1 \) and \( e \rightarrow e_2 \)

\[ \exists e_3 \text{ s.t. } e_1 \rightarrow e_3 \text{ and } \text{Info}[e_2] \leq \text{Info}[e_3] \].