Some more thoughts on $\lambda_{let}$

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

September 21, 2006

Why $\lambda_{let}$ Calculus?

Programming without (recursive) let blocks is tedious

Recursive let blocks can be translated into the $\lambda$-calculus with constants and Y combinator but the translation is

- is complicated (not simple syntactical substitutions);

- is not intuitive or illustrative

- does not match any implementation
  \[ \Rightarrow \text{extend the } \lambda\text{-calculus with recursive let blocks.} \]
\( \lambda \)-calculus with Constants & Letrec

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

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

Statements
\[
S ::= \varepsilon \mid x = E \mid S; S
\]

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

Issues in giving semantics for lets- 1

1. Creating redexes

\[
((\text{let } S \text{ in } \lambda x. e_1) \ e_2)
\]

How do we juxtapose

\[
(\lambda x. e_1) \ e_2
\]

Solution: Lifting rules
Issues in giving semantics for lets- 2

2. How to refer to a variable binding

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

How and when \( f \) and \( y \) refer to their definitions

\[
((\lambda x. e_1) \ y) + y ?
\]

Solution: Instantiation rules

How to define the operational semantics of let blocks: Environments

\[
\text{Eval } [[e]] \rho
\]

An environment-based interpreter.
- An environment where all the (variable name, value) bindings are kept and is passed around for expression evaluation
- When a let expression is encountered the environment is extended with all the let-bindings. Very complicated if the environment contains unevaluated expressions
- Not abstract enough – too many concrete data structures and associated functions for proper execution
How to define the operational semantics of let blocks: graphs

A let simply represents a wiring diagram, i.e., a graph

\[
\text{let} \quad f = \lambda x.e_1 \\
\text{in} \quad y = e_2 \ e_3 \\
\text{let} \quad (f \ y) + y
\]

- Quite complicated to explain \(\beta\)-substitution in a graph based interpreter

How to define the operational semantics of let blocks: via a calculus

- Rewrite rules
  - Lifting rules
  - Instantiation rules (need some new way of writing rules): contexts, ...
- Reduction Strategy
- Normal forms? Equivalences?

let \( x = 5 \) in \( x \)

let \( y = 6 \) in \( y \)

let \( x = 5 \) in \( 5 \)

the \( \lambda_{let} \) calculus
Lifting Rules

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

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

\(\lambda_{let}\) Instantiation Rules

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

Instantiation rule 1

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

Instantiation rule 2

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

Instantiation rule 3

\[
x = a \quad \rightarrow \quad x = C'[C[x]]
\]

where \(a = C[x]\)
Once we have lets we use them elsewhere too ...

The normal $\beta$-rule

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

is replaced 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