A $\lambda$-calculus with Let-blocks (continued)

Arvind
Laboratory for Computer Science
M.I.T.

September 18, 2002

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

Outline

- The $\lambda_{let}$ Calculus
- Some properties of the $\lambda_{let}$ Calculus
λ-calculus with Letrec

E ::= x | λx.E | E E
    | Cond (E, E, E)
    | PFk(E1,...,Ek)
    | CN0
    | CNk(E1,...,Ek) | CNk(SE1,...,SEk)
    | let S in E

PF1 ::= negate | not | ... | Proj1 | Proj2 | ...
PF2 ::= + | ...
CN0 ::= Number | Boolean
CN1 ::= cons | ...

Statements
S ::= ε | x = E | S; S

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

Let-block Statements

" ; " is associative and commutative

S₁ ; S₂ ≡ S₂ ; S₁
S₁ ; (S₂ ; S₃) ≡ (S₁ ; S₂) ; S₃
ε ; S ≡ S
let ε in E ≡ E
Free Variables of an Expression

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(E_1 E_2) &= FV(E_1) \cup FV(E_2) \\
FV(\lambda x. E) &= FV(E) - \{x\} \\
FV(\text{let } S \text{ in } E) &= FVS(S) \cup FV(E) - BVS(S)
\end{align*}
\]

\[
\begin{align*}
FVS(\varepsilon) &= \emptyset \\
FVS(x = E; S) &= FV(E) \cup FVS(S)
\end{align*}
\]

\[
\begin{align*}
BVS(\varepsilon) &= \emptyset \\
BVS(x = E; S) &= \{x\} \cup BVS(S)
\end{align*}
\]

\[\alpha\text{-Renaming (to avoid free variable capture)}\]

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

\[
\begin{align*}
\lambda x.e & \equiv \lambda t.(e[t/x]) \\
\text{let } x = e ; S \text{ in } e_0 & \equiv \text{let } t = e[t/x] ; S[t/x] \text{ in } e_0[t/x]
\end{align*}
\]

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

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

\[
\begin{align*}
\varepsilon[t/x] &= \varepsilon \\
(y = E)[t/x] &= (y = E[t/x]) \\
(S_1 ; S_2)[t/x] &= ? (S_1[t/x] ; S_2[t/x])
\end{align*}
\]
Primitive Functions and Datastructures

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

\( Cond \)-rules
\[ Cond(True, e_1, e_2) \rightarrow e_1 \]
\[ Cond(False, e_1, e_2) \rightarrow e_2 \]

Data-structures
\[ CN_k(e_1, \ldots, e_k) \rightarrow \]
\[ \text{let } t_1 = e_1; \ldots; t_k = e_k \]
\[ \text{in } CN_k(t_1, \ldots, t_k) \]
\[ Prj_i(CN_k(a_1, \ldots, a_k)) \rightarrow a_i \]

The \( \beta \)-rule

The normal \( \beta \)-rule
\[ (\lambda x.e) e_a \rightarrow \emptyset [e_a/x] \]

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 \textit{Instantiation rules} which are used to refer to the value of a variable
Values and Simple Expressions

Values

\[ V ::= \lambda x.E \mid CN_0 \mid CN_k(\text{SE}_1, \ldots, \text{SE}_k) \]

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:

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

Statement Context for an expression

\[ SC[] ::= x = C[] \mid SC[] ; S \mid S ; SC[] \]
\[\lambda_{let} \text{ Instantiation Rules}\]

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

**Instantiation rule 1**

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

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

**Instantiation rule 2**

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

**Instantiation rule 3**

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

---

**Lifting Rules: Motivation**

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

*How do we juxtapose*

\[
(\lambda_x.e_1) \ a \\
or \\
(\lambda_x.e_2) \ e_3 \\
?
\]
Lifting Rules

(\textit{let }S' \textit{in }e') \textit{is the }\alpha?\textit{renamed (let }S \textit{in }e) \textit{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*}

Outline

\begin{itemize}
  \item The \(\lambda_{let}\) Calculus \(\checkmark\)
  \item Some properties of the \(\lambda_{let}\) Calculus \(\leftarrow\)
\end{itemize}
Confluence and Letrecs

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

\textit{substitute for even (n-1) in M}

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

\textit{substitute for odd (n-1) in M}

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

\textit{Can odd in } M_1 \text{ and } M_2 \text{ be reduced to the same expression?}

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

Terms of the $\lambda_{let}$ 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 $\lambda_{let}$ then does there exist a reduction such that $T[[e]] \rightarrow T[[e_1]]$ in $\lambda$?
Instantaneous Information

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

\[ T_P ::= \bot \mid \lambda \ x \ . \ E \mid CN_0 \mid CN_k(T_{P_1}, \ldots, T_{P_k}) \]

Info: \[ E \rightarrow T_P \]

- \[ \text{Info}[\{S \ in \ E\}] = \text{Info}[E] \]
- \[ \text{Info}[\lambda x . E] = \lambda \]
- \[ \text{Info}[CN_0] = CN_0 \]
- \[ \text{Info}[CN_k(a_1, \ldots, a_k)] = CN_k(\text{Info}[a_1], \ldots, \text{Info}[a_k]) \]
- \[ \text{Info}[E] = \bot \quad \text{otherwise} \]

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

Reduction and Info

Terms can be compared by their Info value

- \[ \bot \leq t \quad \text{(bottom)} \]
- \[ t \leq t \quad \text{(reflexive)} \]
- \[ CN_k(v_1, \ldots, v_i, \ldots, v_k) \leq CN_k(v_1, \ldots, v'_i, \ldots, v_k) \quad \text{if } v_i \leq v'_i \]
Print: Unwinding of a term

Print : \( E \rightarrow \{ T_p \} \)
Unwind a term as much as possible using the following instantiation rule (Inst):
\[
(let x = v; S \ in \ C[x]) \rightarrow ?(let x = v; S \ in \ C[v])
\]
and keep track of all the unwindings

Print\[e\] = \{Info\[e_1\] | e \(\rightarrow\) e_1 using the Inst rule\}
Terms with infinite unwindings lead to infinite sets.

Garbage Collection

Let-blocks often contain bindings that are not reachable from the return expression, e.g.,
\[
let x = e \ in \ 5
\]
Such bindings can be deleted without affecting the “meaning” of the term.

**GC-rule**
\[
(let \ S_G; \ S \ in \ e) \rightarrow (let \ S \ in \ e)
\]
provided \( \forall \ x.(x \in (FV(e) \cup FVS(S)) \Rightarrow x \notin BVS(S_G)) \)
Unrestricted Instantiation

\( \lambda_{\text{let}} \) instantiation rules allow only values & variables to be substituted. Let \( \lambda_0 \) be a calculus that permits substitution of arbitrary expressions:

**Unrestricted Instantiation Rules of \( \lambda_0 \)**

\[
\begin{align*}
\text{let } x = e; \ S & \rightarrow \\text{let } x = e; \ S \ \text{in } C[e] \\
(x = e; \ SC[x]) & \rightarrow (x = e; \ SC'[e]) \\
x = e & \rightarrow x = C'[e] \ \text{where } e \equiv C[x]
\end{align*}
\]

Is \( \lambda_0 \) more expressive than \( \lambda_{\text{let}} \) ?

Semantic Equivalence

- What does it mean to say that two terms are equivalent?

- Do any of the following equalities imply semantic equivalence of \( e_1 \) and \( e_2 \)

Syntactic equality of \( \alpha \)-convertability: \( e_1 = e_2 \)

Print equality: \( \text{Print}(e_1) = \text{Print}(e_2) \)

No observable difference in any context: