A λ-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 λ_{let} Calculus
- Some properties of the λ_{let} Calculus

λ-calculus with Letrec

$$
\begin{aligned}
E ::= \; & x \;|\; \lambda x.E \;|\; E\; E \\
& |\; \text{Cond}\,(E, E, E) \\
& |\; PF_k(E_1,\dots,E_k) \\
& |\; CN_0 \\
& |\; CN_k(E_1,\dots,E_k) \;|\; \underline{CN}_k(SE_1,\dots,SE_k) \leftarrow \textit{not in initial terms} \\
& |\; \textit{let}\; S \;\textit{in}\; E
\end{aligned}
$$

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

$$
\begin{aligned}
S_1 \;;\; S_2 \quad & \equiv S_2 \;;\; S_1 \\
S_1 \;;\; (S_2 \;;\; S_3) \quad & \equiv (S_1 \;;\; S_2\;) \;;\; S_3 \\
\\
\varepsilon \;;\; S \quad & \equiv S \\
\textit{let}\; \varepsilon \;\textit{in}\; E \quad & \equiv E
\end{aligned}
$$

Free Variables of an Expression

FV(x) = {x}
FV(E_1 E_2) = FV(E_1) U FV(E_2)
FV(λx.E) = FV(E) - {x}
FV(*let* S *in* E) = FVS(S) U FV(E) – BVS(S)

FVS(ε) = {}
FVS(x = E; S)= FV(E) U FVS(S)

BVS(ε) = {}
BVS(x = E; S)= {x} U BVS(S)

α - Renaming *(to avoid free variable capture)*

Assuming t is a new variable, rename x to t :
λx.e $\equiv$ λt.(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 [t/x] is defined as follows:

x[t/x] = t
y[t/x] = y *if* x $\neq$ y
(E_1 E_2)[t/x] = (E_1[t/x] E_2[t/x])
(λx.E)[t/x] = λx.E
(λy.E)[t/x] = ?y.E[t/x] *if* x $\neq$ y
(*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)

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])

Primitive Functions and Datastructures

δ-rules

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

...

Cond-rules

$\mathrm{Cond}(\mathrm{True}, e_1, e_2) \rightarrow e_1$?

$\mathrm{Cond}(\mathrm{False}, e_1, e_2) \rightarrow e_2$

Data-structures

$CN_k(e_1,\ldots,e_k) \rightarrow$

$\quad$ *let* $t_1 = e_1; \ldots ; t_k = e_k$

$\quad$ *in* $\underline{CN_k}(t_1,\ldots,t_k)$

$Prj_i(\underline{CN_k}(a_1,\ldots,a_k)) \rightarrow a_i$

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 to refer to the value of a variable

Values and Simple Expressions

Values

V ::= λx.E | CN$_0$ | $\underline{\text{CN}_k}$(SE$_1$,...,SE$_k$)

Simple expressions

SE ::= x | 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[] ::= []
| 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[]

λ_{let} Instantiation Rules

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

Instantiation rule 1

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

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

$x = a \qquad \rightarrow x = C'[C[x]] \qquad \textit{where } a = C[x]$

Lifting Rules: Motivation

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

How do we juxtapose

$(\lambda x.e_1)\ \ a$

or

$(\lambda x.e_2)\ e_3$?

Lifting Rules

(*let* S’ *in* e’) is the α?*renamed* (*let* S *in* e) to avoid name conflicts in the following rules:

x = *let* S *in* e → x = e’; S’

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

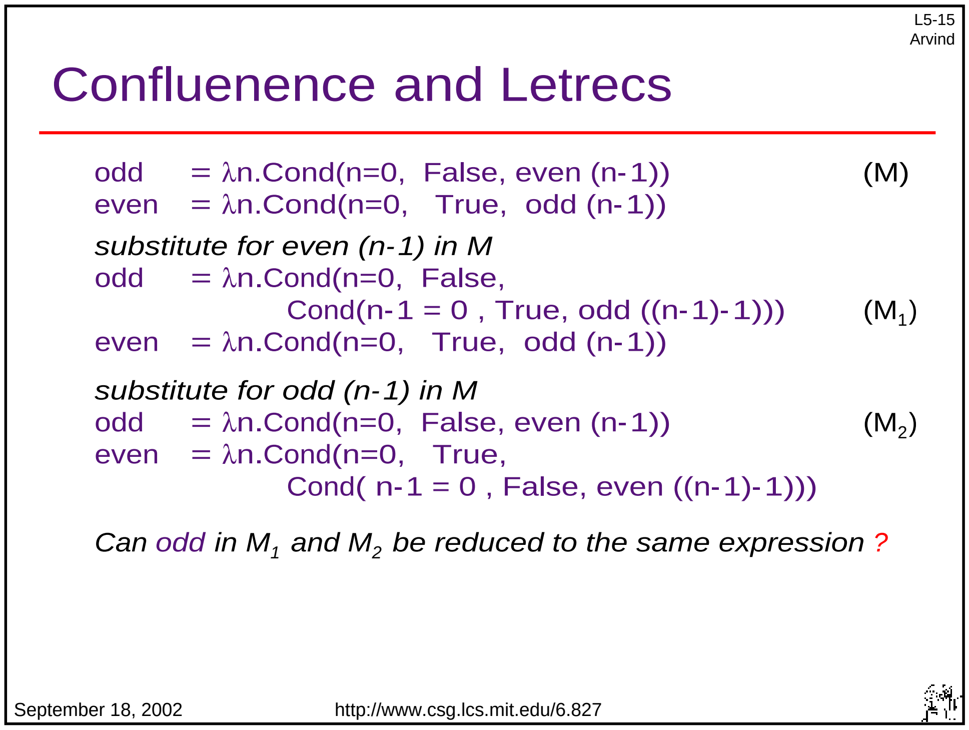
(*let* S *in* e) e_1 → *let* S’ *in* e’ e_1

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)

Outline

- The λ_{let} Calculus √
- Some properties of the λ_{let} Calculus ←



L5-15
Arvind

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

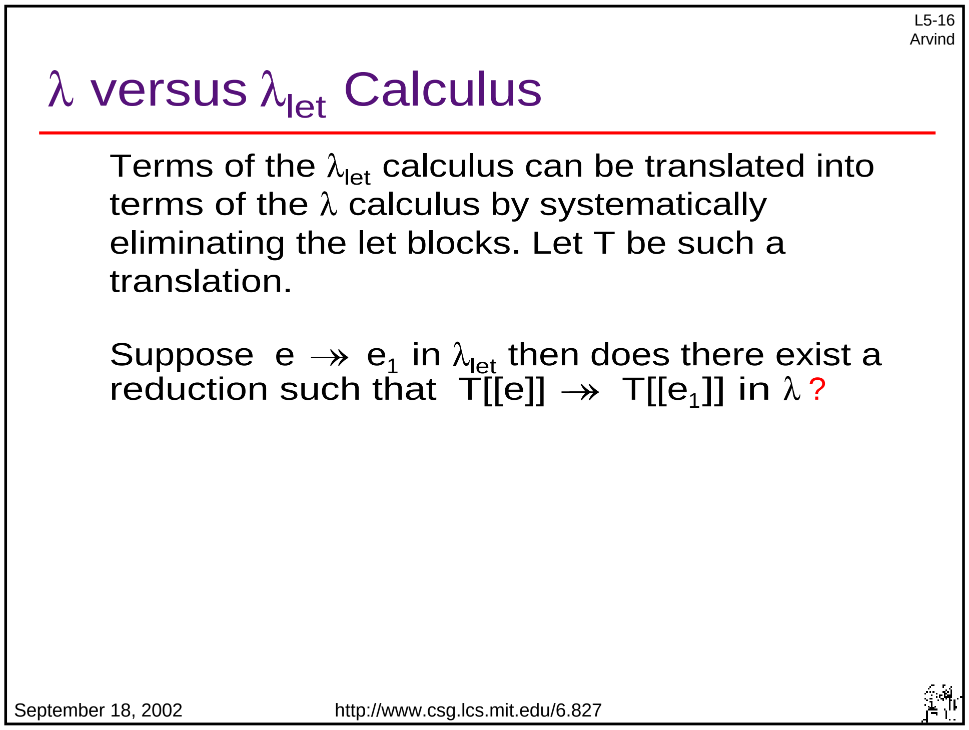
```
odd    = λn.Cond(n=0,  False,
                Cond(n-1 = 0 , True, odd ((n-1)-1)))       (M1)
even   = λn.Cond(n=0,   True,  odd (n-1))
```

substitute for odd (n-1) in M

```
odd    = λn.Cond(n=0,  False, even (n-1))                  (M2)
even   = λn.Cond(n=0,   True,
                Cond( n-1 = 0 , False, even ((n-1)-1)))
```

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

September 18, 2002 http://www.csg.lcs.mit.edu/6.827



L5-16
Arvind

λ versus λ_{let} Calculus

Terms of the λ_{let} calculus can be translated into terms of the λ calculus by systematically eliminating the let blocks. Let T be such a translation.

Suppose $e \twoheadrightarrow e_1$ in λ_{let} then does there exist a reduction such that $T[[e]] \twoheadrightarrow T[[e_1]]$ in λ ?

September 18, 2002 http://www.csg.lcs.mit.edu/6.827

Instantaneous Information

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

$T_P \quad ::= \quad \perp \mid \lambda \mid CN_0 \mid CN_k(T_{P1},\ldots,T_{Pk})$

Info: $\quad E \rightarrow T_P$

$$
\begin{array}{lll}
\text{Info}[\{S \textit{ in } E\}] & = & \text{Info }[E] \\
\text{Info}[\lambda x.E] & = & \lambda \\
\text{Info}[CN_0] & = & CN_0 \\
\text{Info}[\underline{CN}_k(a_1,\ldots,a_k)] & & \\
 & = & CN_k(\text{Info}[a_1],\ldots,\text{Info}[a_k]) \\
\text{Info}[E] & = & \perp \qquad\qquad \textit{otherwise}
\end{array}
$$

Reduction and Info

Terms can be compared by their Info value

$$
\boxed{
\begin{array}{llll}
\perp & \leq & t & \textit{(bottom)} \\
t & \leq & t & \textit{(reflexive)} \\
CN_k(v_1,\ldots,v_i,\ldots,v_k) & \leq & CN_k(v_1,\ldots,v'_i,\ldots,v_k) & \\
 & & \textit{if} \quad v_i \leq v'_i &
\end{array}
}
$$

Proposition Reduction is monotonic wrt Info:
If $e \twoheadrightarrow e_1$ then $\text{Info}[e] \leq \text{Info}[e_1]$.

Proposition Confluence wrt Info:
If $e \twoheadrightarrow e_1$ and $e \twoheadrightarrow e_2$ then
$\exists\ e_3$ s.t. $e_1 \twoheadrightarrow e_3$ and $\text{Info}[e_2] \leq \text{Info}[e_3]$.

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]) → ?(*let* x = v; S *in* C[v])

and keep track of all the unwindings

Print[e] = {Info[e_1] | e ↠ 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) → (*let* S *in* e)

provided $\forall$ x.(x $\in$ (FV(e) U FVS(S)) $\Rightarrow$ x $\notin$ BVS(S_G))

Unrestricted Instantiation

λ_{let} instantiation rules allow only values & variables to be substituted. Let λ_0 be a calculus that permits substitution of arbitrary expressions:

Unrestricted Instantiation Rules of λ_0

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

Is λ_0 more expressive than λ_{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 α-convertability: $e_1 = e_2$

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

No observable difference in any context: