Some more thoughts on λ_{let}

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

September 21, 2006

Why λ_{let} Calculus ?

Programming without (recursive) *let* blocks is tedious

Recursive *let* blocks can be translated into the λ-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
  - $\Rightarrow$ extend the λ-calculus with *recursive let blocks.*

λ-calculus with Constants & Letrec

$E ::= x \mid \lambda x.E \mid E\ E$
$\mid Cond\ (E, E, E)$
$\mid PF_k(E_1,\ldots,E_k)$
$\mid CN_0$
$\mid CN_k(E_1,\ldots,E_k) \mid \underline{CN_k}(SE_1,\ldots,SE_k)$ *not in initial terms*
$\mid$ *let* S *in* E

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

Issues in giving semantics for lets- 1

1. Creating redexes

((*let* S *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

let

$f = \lambda x.e_1$

$y = e_2\ e_3$

in

$(f\ y) + y$

Solution: Instantiation rules

How and when f and y refer to their definitions

$((\lambda x.e_1)\ \ y) + y$?

How to define the operational semantics of let blocks: *Environments*

Eval [[e]] ρ

environment

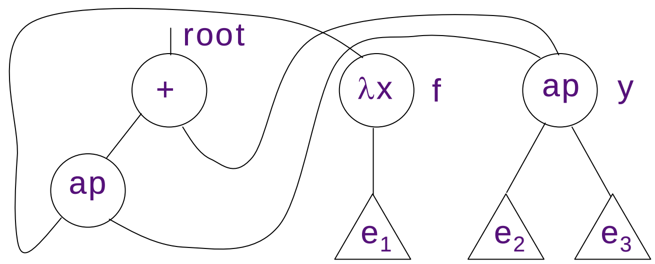
An environment-based interpreter.

- 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

let
    $f = \lambda x.e_1$
    $y = e_2 \; e_3$
in
    (f y) + y



– Quite complicated to explain β-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 x = 5 in 5	5	let x = 5 y = 6 in x

the λ_{let} calculus

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)

λ_{let} 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]) → (*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]) → (x = a ; SC'[a])

Instantiation rule 3
x = a → x = C'[C[x]]
where a = C[x]

Once we have lets we use them elsewhere too ...

The normal β-rule

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

is replaced the following β-rule

$(\lambda x.e)\ e_a \rightarrow \textit{let}\ t = e_a\ \textit{in}\ e[t/x]$
where t is a new variable

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