Starting Point: Standard CBV λ-calculus. It's No Good

The usual CBV strategy

$$\begin{array}{lllll} \text{(app)} & (\lambda z{:}T.e')v & \longrightarrow & \{v/z\}e' \\ \text{(let)} & \textbf{let}\ \ z = v\ \ \textbf{in}\ \ e & \longrightarrow & \{v/z\}e \end{array}$$

loses too much information. For example, in (let), if

- e contains **send** $machineB\ \ (...z...)$, and we want z to be rebound to a local resource v' at $machineB$; or
- we dynamically update the z binding after the (let) step.

So first explore refined strategies with *delayed instantiation* – but stay 'essentially' CBV. Then add dynamic rebinding (or update).

Three CBV λ-calculi

- λ_c *construct-time* (the standard one) – instantiate identifiers as soon as they are bound to values
- λ_r *redex-time* – instantiate identifiers when they appear in redex position
- λ_d *destruct-time* – instantiate identifiers only when under destructors

Then:

- λ_{marsh} = destruct-time lambda + *marshalling* (with rebinding)
- $\lambda_{\mathrm{update}}$ = destruct-time lambda + *dynamic update*

Delayed Instantiation Examples

	Construct-time λ_c	Redex-time λ_r	Destruct-time λ_d
	$(\lambda z.7)8$	$(\lambda z.7)8$	$(\lambda z.7)8$
$\longrightarrow$	7	$\textbf{let } z = 8 \textbf{ in } 7$	$\textbf{let } z = 8 \textbf{ in } 7$
	$\textbf{let } x = 5 \textbf{ in } \pi_1\,(x, x)$	$\textbf{let } x = 5 \textbf{ in } \pi_1\,(x, x)$	$\textbf{let } x = 5 \textbf{ in } \pi_1\,(x, x)$
$\longrightarrow$	$\pi_1\,(5, 5)$	$\textbf{let } x = 5 \textbf{ in } \pi_1\,(5, x)$	$\textbf{let } x = 5 \textbf{ in } x$
$\longrightarrow$	5	$\textbf{let } x = 5 \textbf{ in } \pi_1\,(5, 5)$	
$\longrightarrow$		$\textbf{let } x = 5 \textbf{ in } 5$	

Redex-time Semantics

Values: $u ::= \ldots \mid \textbf{let } z = u \textbf{ in } u$

Reduction contexts E_3:

the usual evaluation contexts, and also value-lets $\textbf{let } z = u \textbf{ in } \_$

Let E_2 range over pure value-let contexts.

(proj)	$\pi_r\,(E_2.(u_1, u_2))$	$\longrightarrow$	$E_2.u_r$	
(app)	$(E_2.(\lambda z{:}T.e))u$	$\longrightarrow$	$E_2.\textbf{let } z = u \textbf{ in } e$	if ...
(inst)	$\textbf{let } z = u \textbf{ in } E_3.z$	$\longrightarrow$	$\textbf{let } z = u \textbf{ in } E_3.u$	if ...

($E.e$) (*don't* substitute, instead instantiate single occurrences) (**letrec** too)

Destruct-time Semantics

similar, except (1) values include x, and (2) instantiate only variables under *destruct contexts*

$$R \quad ::= \quad \pi_r \ \_ \mid \_ \ u$$

Examples:

$\textbf{let } x = (1, 2) \textbf{ in } \pi_1 \ x$	$\pi_1 \, (\textbf{let } x = 1 \textbf{ in } (x, x))$

Destruct-time Semantics

similar, except (1) values include x, and (2) instantiate only variables under *destruct contexts*

$$R \quad ::= \quad \pi_r \; \_ \mid \_ \; u$$

Examples:

$\quad \textbf{let } x = (1,2) \textbf{ in } \pi_1 \; x$	$\quad \pi_1 \, (\textbf{let } x = 1 \textbf{ in } (x, x))$
$\longrightarrow \quad \textbf{let } x = (1,2) \textbf{ in } \pi_1 \, (1,2)$	$\longrightarrow \quad \textbf{let } x = 1 \textbf{ in } x$
$\longrightarrow \quad \textbf{let } x = (1,2) \textbf{ in } 1$	$\not\longrightarrow$

Theorems

- Sanity (Unique decomposition, Type soundness)
- Redex- and Destruct-time are still CBV (w.r.t. observational congruence — use bisimulation proof techniques. Stoyle)

Now we have delayed instantation semantics, we can add marshalling primitives (later: update too).

Here work with destruct-time (so more rebinds/updates take effect). Later (in Acute) use redex-time – debatable...

Dynamic Rebinding: λ_{marsh} Constructs

(Ultimately want distributed comms, but λ is enough)

Take λ_d and add constructs to *mark* contexts

$$e ::= \ldots \mid \textbf{mark}\ \ M\ \ \textbf{in}\ \ e$$

where M is a *mark name* (this is not a binder), and to *package* and *unpackage* values

$$e ::= \ldots \mid \textbf{marshal}\ M\ \ e \mid \textbf{unmarshal}\ M\ \ e$$

which are both with respect to a mark.

λ_{marsh}: Example

Marks are used to specify which variables get rebound:

$$
\begin{array}{l}
\textbf{let}\ \ y_1{:}\mathsf{int} = 6\ \ \textbf{in} \\
\textbf{mark}\ \ M\ \ \textbf{in} \\
\textbf{let}\ \ x_1{:}\mathsf{Marsh}\ (\mathsf{int} * \mathsf{int}) = \\
\quad \textbf{let}\ \ z_1{:}\mathsf{int} = 3\ \ \textbf{in} \\
\quad \textbf{marshal}\ M\ (z_1, y_1) \\
\textbf{in} \\
\textbf{let}\ \ y_2{:}\mathsf{int} = 7\ \ \textbf{in} \\
\textbf{mark}\ \ M'\ \ \textbf{in} \\
\textbf{unmarshal}\ M'\ x_1
\end{array}
$$

$$\xrightarrow{\text{(marshal)}}$$

$$
\begin{array}{ll}
\textbf{let } y_1\text{:int} = 6 \textbf{ in} & \textbf{let } y_1\text{:int} = 6 \textbf{ in} \\
\textbf{mark } M \textbf{ in} \;\ldots\ldots\ldots\ldots\ldots & \textbf{mark } M \textbf{ in} \\
\textbf{let } x_1\text{:Marsh (int} * \text{int)} = (& \textbf{let } x_1 : \ldots = (\\
\quad \textbf{let } z_1\text{:int} = 3 \textbf{ in} & \quad \textbf{let } z_1\text{:int} = 3 \textbf{ in} \\
\quad \boxed{\textbf{marshal } M \; (z_1, y_1))} \textbf{ in} & \quad \boxed{\begin{array}{l}\textbf{marshalled } (y_0\text{:int}) \; (\\ \quad \textbf{let } z_1\text{:int} = 3 \textbf{ in} \\ \quad (z_1, y_0))) \textbf{ in}\end{array}} \\
\textbf{let } y_2\text{:int} = 7 \textbf{ in} & \textbf{let } y_2\text{:int} = 7 \textbf{ in} \\
\textbf{mark } M' \textbf{ in} & \textbf{mark } M' \textbf{ in} \\
\textbf{unmarshal } M' \; x_1 & \textbf{unmarshal } M' \; x_1
\end{array}
$$

let y_1:int $= 6$ **in**

mark M **in**

$\boxed{\textbf{let}\ \ x_1 : \ldots = (}$

  let z_1:int $= 3$ **in**

  marshalled (y_0:int) (

    let z_1:int $= 3$ **in**

    (z_1, y_0))) **in**

let y_2:int $= 7$ **in**

mark M' **in**

unmarshal M' $\boxed{x_1}$

$\xrightarrow{\text{(inst)}}$

let y_1:int $= 6$ **in**

mark M **in**

let $x_1 : \ldots = ($

  let z_1:int $= 3$ **in**

  marshalled (y_0:int) (

    let z_1:int $= 3$ **in**

    (z_1, y_0))) **in**

let y_2:int $= 7$ **in**

mark M' **in**

unmarshal M' (

$$\boxed{\begin{array}{l}\textbf{let}\ \ z_1\text{:int} = 3\ \ \textbf{in}\\ \textbf{marshalled}\ (y_0\text{:int})\ \ (\\ \quad\textbf{let}\ \ z_1\text{:int} = 3\ \ \textbf{in}\\ \quad(z_1, y_0)))\end{array}}$$

$$\xrightarrow{\text{(unmarshal)}}$$

$\textbf{let}\ \ y_1{:}\mathsf{int} = 6\ \ \textbf{in}$

$\textbf{mark}\ \ M\ \ \textbf{in}$

$\textbf{let}\ \ x_1{:}\mathsf{Marsh}\ (\mathsf{int} * \mathsf{int}) = \ldots\ \textbf{in}$

$\textbf{let}\ \ y_2{:}\mathsf{int} = 7\ \ \textbf{in}$

$\textbf{mark}\ \ M'\ \ \textbf{in}$ . . . .

$\boxed{\textbf{unmarshal}\ M'}\ ($

$\quad\textbf{let}\ \ z_1{:}\mathsf{int} = 3\ \ \textbf{in}$

$\quad\textbf{marshalled}\ (y_0{:}\mathsf{int})\ \ ($

$\qquad\textbf{let}\ \ z_1{:}\mathsf{int} = 3\ \ \textbf{in}$

$\qquad(z_1, y_0)))$

$\textbf{let}\ \ y_1{:}\mathsf{int} = 6\ \ \textbf{in}$

$\textbf{mark}\ \ M\ \ \textbf{in}$

$\textbf{let}\ \ x_1{:}\mathsf{Marsh}\ (\mathsf{int} * \mathsf{int}) = \ldots\ \textbf{in}$

$\textbf{let}\ \ y_2{:}\mathsf{int} = 7\ \ \textbf{in}$

$\textbf{mark}\ \ M'\ \ \textbf{in}$ . . . .

$\textbf{let}\ \ z_1{:}\mathsf{int} = 3\ \ \textbf{in}$

(z_1, y_2)

λ_{marsh}: Design Points

Direct semantics, without encoding.

Need external/internal names (ext. meaningful between programs).

Only one sort of variable, not 'dynamic' and 'static'.

No dynamic rebinding except at **unmarshal**-time.

No runtime errors except at **unmarshal**-time (with simple typing).

marshal reminiscent of **call/cc** – but here pulling out the binding part of the context, there the evaluation part.