Dynamic Rebinding for Marshalling and Update, with Destruct-time $\lambda$

Gavin Bierman$^\dagger$ Michael Hicks$^\ddagger$ Peter Sewell$^\dagger$
Gareth Stoyle$^\dagger$ Keith Wansbrough$^\dagger$

$^\dagger$University of Cambridge
{First.Last}@cl.cam.ac.uk

$^\ddagger$University of Maryland
mwh@cs.umd.edu
Dynamic Binding – Why?

Static binding good, dynamic binding bad. But, need:

- Dynamic Linking
- Dynamic Rebinding for marshalled values
- Dynamic Update for long-running systems (c.f. Erlang)

going to show some core mechanisms, with clean reduction semantics.

View as steps towards design of ML-like languages for distributed computation.
Dynamic Rebinding – Marshalling Scenarios

Consider sending a value (a thunk) between machines.

It may contain identifiers for:

1. ubiquitous standard library functions – should be rebound
2. application-specific location-dependent libraries – should be rebound
3. other let-bound application values – which should be sent with it

Further, may want to rebind to non-standard definitions, to securely encapsulate (sandbox) untrusted code.
Starting Point: Standard CBV $\lambda$-calculus. It’s No Good

The usual CBV strategy

\[
\begin{align*}
\text{(app)} & \quad (\lambda z : T. e')v & \rightarrow & \{v/z\}e' \\
\text{(let)} & \quad \text{let } z = v \text{ in } e & \rightarrow & \{v/z\}e
\end{align*}
\]

loses too much information, eg in (let) if

- $e$ sends a value mentioning $z$ to another machine, and we want $z$ to be rebound to a local resource; 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 and update.
Three CBV $\lambda$-calculi

- $\lambda_c$ *construct-time* (the standard one) – instantiate identifiers as soon as they are bound to values
- $\lambda_r$ *redex-time* – instantiate identifiers when they appear in redex position
- $\lambda_d$ *destruct-time* – instantiate identifiers only when under destructors
Examples (1), (2)

<table>
<thead>
<tr>
<th>Construct-time $\lambda_c$</th>
<th>Redex-time $\lambda_r$</th>
<th>Destruct-time $\lambda_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\lambda z . 7) 8$</td>
<td>$(\lambda z . 7) 8$</td>
<td>$(\lambda z . 7) 8$</td>
</tr>
<tr>
<td>$7$</td>
<td>let $z = 8$ in $7$</td>
<td>let $z = 8$ in $7$</td>
</tr>
<tr>
<td>let $x = 5$ in $\pi_1(x, x)$</td>
<td>let $x = 5$ in $\pi_1(x, x)$</td>
<td>let $x = 5$ in $\pi_1(x, x)$</td>
</tr>
<tr>
<td>$\pi_1(5, 5)$</td>
<td>let $x = 5$ in $\pi_1(5, x)$</td>
<td>let $x = 5$ in $x$</td>
</tr>
<tr>
<td>$5$</td>
<td>let $x = 5$ in $\pi_1(5, 5)$</td>
<td>let $x = 5$ in $5$</td>
</tr>
</tbody>
</table>
Redex-time Semantics

Reduction contexts: Reduce under standard evaluation contexts, but also under value-lets \textbf{let } z = u \textbf{ in } _

Values include \textbf{let } z = u \textbf{ in } u

value-let binding contexts \( E_2 \)

mixed value-let and evaluation contexts \( E_3 \)

\begin{align*}
\text{(proj)} & \quad \pi_r(E_2.(u_1, u_2)) \quad \rightarrow \quad E_2.u_r \\
\text{(app)} & \quad (E_2.(\lambda z:T.e))u \quad \rightarrow \quad E_2.\textbf{let } z = u \textbf{ in } e \quad \text{if } \ldots \\
\text{(inst)} & \quad \textbf{let } z = u \textbf{ in } E_3.z \quad \rightarrow \quad \textbf{let } z = u \textbf{ in } E_3.u \quad \text{if } \ldots
\end{align*}

Don’t substitute, instead instantiate single occurrences
Destruct-time Semantics

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

$$R ::= \pi_{r-} \mid _u$$

Properties

- Sanity
- Redex- and Destruct time are still CBV
Dynamic Rebinding: $\lambda_{\text{marsh}}$ Constructs

(Ultimately want distributed comms, but $\lambda$ is enough)

Take $\lambda_d$ and add constructs to mark contexts

$$e ::= ... \mid \text{mark } M \text{ in } e$$

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

$$e ::= ... \mid \text{marshal } M \ e \mid \text{unmarshal } M \ e$$

which are both with respect to a mark.
\[ \lambda_{\text{marsh}}: \text{Example} \]

Marks are used to specify which variables get rebound

\[
\begin{align*}
\text{let } y_1 : \text{int} &= 6 \text{ in} \\
\text{mark } M \text{ in} \\
\text{let } x_1 : \text{Marsh (int} \times \text{int}) &= ( \\
    \text{let } z_1 : \text{int} &= 3 \text{ in} \\
    \text{marshal } M (y_1, z_1) \text{ in} \\
\text{let } y_2 : \text{int} &= 7 \text{ in} \\
\text{mark } M' \text{ in} \\
\text{unmarshal } M' x_1
\end{align*}
\]
let $y_1$:int = 6 in
mark $M$ in
let $x_1$:Marsh (int * int) = (let $z_1$:int = 3 in marshal $M$ ($y_1$, $z_1$)) in
let $y_2$:int = 7 in
mark $M'$ in
unmarshal $M'$ $x_1$

let $y_1$:int = 6 in
mark $M$ in
let $x_1$: $T$ = (let $z_1$:int = 3 in
marshalled ($y_0$:int) (let $z_1$:int = 3 in ($y_0$, $z_1$))) in
let $y_2$:int = 7 in
mark $M'$ in
unmarshal $M'$ $x_1$
let $y_1: \text{int} = 6$ in
mark $M$ in
let $x_1: T = ( $
| let $z_1: \text{int} = 3$ in
| marshalled $(y_0: \text{int}) ( $
| | let $z_1: \text{int} = 3$ in
| | $(y_0, z_1)))$ in
| let $y_2: \text{int} = 7$ in
mark $M'$ in
unmarshal $M'$ $x_1$

let $y_1: \text{int} = 6$ in
mark $M$ in
let $x_1: T = ( $
| let $z_1: \text{int} = 3$ in
| marshalled $(y_0: \text{int}) ( $
| | let $z_1: \text{int} = 3$ in
| | $(y_0, z_1)))$ in
| let $y_2: \text{int} = 7$ in
mark $M'$ in
unmarshal $M'$ $( $
let $y_1$::int = 6 in
mark $M$ in
let $x_1$::Marsh (int * int) = ... in
let $y_2$::int = 7 in
mark $M'$ in
unmarshal $M'$ (let $z_1$::int = 3 in
marshalled ($y_0$::int) (let $z_1$::int = 3 in
($y_0$, $z_1$)))

$\rightarrow$

let $y_1$::int = 6 in
mark $M$ in
let $x_1$::Marsh (int * int) = ... in
let $y_2$::int = 7 in
mark $M'$ in
let $z_1$::int = 3 in
(y_2, z_1)$
\( \lambda_{\text{marsh}} : \text{Semantics} \)

Use destruct-time lambda, plus rules for \textit{marshal} and \textit{unmarshal}. 
Consider systems that must provide uninterrupted service. They must be *dynamically updated* to fix bugs and add new functionality.

Many forms of update are possible. Several systems have been built, but there is little semantics.

Here, show how a simple (but already expressive) form of update to CBV functional programs can be based on $\lambda_d$. 
\[ \lambda_{\text{update}}: \text{Example} \]

\[
\begin{align*}
\text{let } x_1 &= 5 \text{ in} & \{y \leftarrow (x_1, 6)\} & \text{let } x_1 &= 5 \text{ in} \\
\text{let } y_1 &= (4, 6) \text{ in} & \text{let } y_1 &= (x_1, 6) \text{ in} \\
\text{let } z_1 &= \text{update} \text{ in} & \text{let } z_1 &= () \text{ in} \\
\pi_1 y_1 & \pi_1 y_1
\end{align*}
\]

Update is synchronous – when \texttt{update} appears in a reduction context.

Any identifier in scope at the update point (here \(x\) or \(y\)) can be rebound, to an expression that may mention any identifiers in scope at its binding point.
Use destruct-time lambda, plus one rule for `update`.
Conclusion

Reasonably nice primitives for often-fudged problems.

Future Directions

Many other issues, in both

- Marshalling
- Update

Paper at http://www.cl.cam.ac.uk/users/pes20
The End