Dynamic Rebinding for Marshalling and Update, with Destruct-time λ

Gavin Bierman[†] **Michael Hicks**[‡] **Peter Sewell**[†]
Gareth Stoyle[†] **Keith Wansbrough**[†]

[†]**University of Cambridge**

{First.Last}@cl.cam.ac.uk

[‡]**University of Maryland**

mwh@cs.umd.edu

Dynamic Binding – Why?

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

- Dynamic Linking
- Dynamic *R*ebinding 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 λ-calculus. It's No Good

The usual CBV strategy

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

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 instantation* – but stay 'essentially' CBV. Then add dynamic rebinding and 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

Examples (1), (2)

	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	**let** $z = 8$ **in** 7	**let** $z = 8$ **in** 7
	let $x = 5$ **in** $\pi_1(x, x)$	**let** $x = 5$ **in** $\pi_1(x, x)$	**let** $x = 5$ **in** $\pi_1(x, x)$
$\longrightarrow$	$\pi_1(5, 5)$	**let** $x = 5$ **in** $\pi_1(5, x)$	**let** $x = 5$ **in** x
$\longrightarrow$	5	**let** $x = 5$ **in** $\pi_1(5, 5)$	
$\longrightarrow$		**let** $x = 5$ **in** 5	

Redex-time Semantics

Reduction contexts: Reduce under standard evaluation contexts, but also under value-lets **let** $z = u$ **in** _

Values include **let** $z = u$ **in** u

value-let binding contexts E_2

mixed value-let and evaluation contexts E_3

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

Don't substitute, instead instantiate single occurrences

Destruct-time Semantics

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

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

Properties

- Sanity
- Redex- and Destruct time are still CBV

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

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

$$\longrightarrow$$

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

mark M **in**

let x_1:Marsh (int $*$ int) $= ($

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

  $\boxed{\textbf{marshal}\ M\ (y_1, z_1))}$ **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**

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

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

mark M' **in**

unmarshal $M'\ \ x_1$

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

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

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

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

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

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

$\qquad (y_0, z_1)))\ \textbf{in}$

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

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

$\textbf{unmarshal} M'\ \boxed{x_1}$

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

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

$\textbf{let}\ x_1{:}T = ($

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

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

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

$\qquad (y_0, z_1)))\ \textbf{in}$

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

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

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

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

$$\longrightarrow$$

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

mark M **in**

let x_1:Marsh (int $*$ int) $= \ldots$ **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**

$(y_0, z_1)))$

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

mark M **in**

let x_1:Marsh (int $*$ int) $= \ldots$ **in**

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

mark M' **in**

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

(y_2, z_1)

λ_{marsh}: Semantics

Use destruct-time lambda, plus rules for **marshal** and **unmarshal**.

Dynamic Update: Scenarios

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 λ_d.

λ_{update}: Example

$$\begin{array}{lcl}
\textbf{let } x_1 = 5 \textbf{ in} & \xrightarrow{\{y \Leftarrow (x_1, 6)\}} & \textbf{let } x_1 = 5 \textbf{ in} \\
\textbf{let } y_1 = (4, 6) \textbf{ in} & & \textbf{let } y_1 = (x_1, 6) \textbf{ in} \\
\textbf{let } z_1 = \textbf{update in} & & \textbf{let } z_1 = () \textbf{ in} \\
\pi_1 y_1 & & \pi_1 y_1
\end{array}$$

Update is synchronous – when **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.

λ_{update}: **Semantics**

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