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, College Park
mwh@cs.umd.edu

Abstract

Most programming languages adopt static binding, but for distributed programming an exclusive reliance on static binding is too restrictive: dynamic binding is required in various guises, for example when a marshalled value is received from the network, containing identifiers that must be rebound to local resources. Typically it is provided only by ad-hoc mechanisms that lack clean semantics.

In this paper we adopt a foundational approach, developing core dynamic rebinding mechanisms as extensions to simply-typed call-by-value λ-calculus. To do so we must first explore refinements of the call-by-value reduction strategy that delay instantiation, to ensure computations make use of the most recent versions of rebound definitions. We introduce *redex-time* and *destruct-time* strategies. The latter forms the basis for a λ_{marsh} calculus that supports dynamic rebinding of marshalled values, while remaining as far as possible statically-typed. We sketch an extension of λ_{marsh} with concurrency and communication, giving examples showing how wrappers for encapsulating untrusted code can be expressed. Finally, we show that a high-level semantics for dynamic updating can also be based on the destruct-time strategy, defining a λ_{update} calculus with simple primitives to provide type-safe updating of running code. We thereby establish primitives and a common semantic foundation for a variety of real-world dynamic rebinding requirements.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms Languages, Theory, Verification

Keywords programming languages, dynamic binding, dynamic update, marshalling, serialisation, distributed programming, lambda calculus

1. Introduction

Most programming languages employ *static binding*, with the meaning of identifiers determined by their compile-time context. In general, this gives more comprehensible code than *dynamic binding* alternatives, where the meanings of identifiers depend in some sense on their 'use-time' contexts; static binding is also a requirement for conventional static type systems. Modern software, though, is becoming increasingly dynamic, as it becomes ever more modular, extensible, and distributed. Exclusive use of static binding is too limiting in many ways:

- When values or computations are marshalled from a running system and moved elsewhere, either by network communication or via a persistent store, some of their identifiers may need to be *dynamically rebound*. These may be both 'external' identifiers of system-calls or language run-time library functions, and, more interestingly, 'internal' identifiers from application libraries which exist in the new context. Such libraries should not be automatically copied with values that use them, both for performance reasons and as they may have location-dependent behaviour (*e.g.*, routing functions). Moreover, a value may be moved repeatedly, and the set of identifiers to be rebound may change as it moves. For example, it may be desirable to acquire an organisation-specific library that, once resolved, should be fixed and carried with code moved within that organisation.

- Flexible control of dynamic rebinding can support *secure encapsulation* of untrusted code, by allowing access only to sandboxed resources. For example, when loading an untrusted applet, we may bind its `open` identifier to a `safe_open` function that only opens files in the `/tmp` directory. On the other hand, we want the flexibility to link trusted code with the unconstrained `open` function.

- Systems that must provide uninterrupted service (*e.g.*, telephone switches) must be *dynamically updated* to fix bugs and add new functionality – essentially by loading new code into the program and then dynamically rebinding some of the existing identifiers to the new definitions.

While dynamic rebinding is clearly useful in practice, most modern programming languages provide only rather limited and ad-hoc mechanisms. Moreover, no adequate semantic understanding of rebinding currently exists. Our goal in this paper is to identify core mechanisms for dynamic rebinding, as a step towards the design of improved languages for distributed computation.

We are focussing on distributed ML-like languages: with higher-order functions, for expressiveness; with call-by-value (CBV) reduction, for a simple evaluation order (desirable in the presence of either communication effects or dynamic updates); and where possible with static typing, as early detection of errors is particularly important in both distributed and long-running systems.

The motivations for dynamic rebinding arise from distribution, but it turns out that the essential problems come from the interaction between rebinding and sequential computation. We therefore begin with the simply-typed CBV lambda-calculus and develop calculi that support rebinding for marshalling and update. To demonstrate feasibility we sketch an extension of the former with inter-machine communication, and discuss a possible implementation.

We express the semantics of these calculi with direct operational semantics, defining reductions over the calculus syntax. This approach provides clarity, and should scale well to full language designs; it avoids commitment to any particular implementation strategy. We find this preferable to the lower-level alternatives of expressing semantics using abstract machines or encodings (into languages with references), which we believe would lead to rather complex definitions.

A full version is available as a technical report [BHS$^+$03].

Revisiting CBV λ-Calculus Consider the CBV λ-calculus, a model fragment of ML, and in particular the way in which identifiers are instantiated. The usual operational semantics substitutes out binders – the standard *construct-time* (app) and (let) rules

$$\begin{array}{lll} \text{(app)} & (\lambda z{:}T.e)v & \longrightarrow & \{v/z\}e \\ \text{(let)} & \textbf{let}\ z = v\ \textbf{in}\ e & \longrightarrow & \{v/z\}e \end{array}$$

instantiate all instances of z as soon as the value v that it has been bound to has been constructed.

This semantics is not compatible with dynamic rebinding, as it loses too much information. To see this, suppose that e in **let** $z = v$ **in** e transmits a function containing z to some other machine, and we have indicated somehow that z should be dynamically rebound to the local definition when it arrives. With the (let) rule this would be futile, as the z is substituted away before the communication occurs. Similarly, a dynamic update of z after a (let) would be vacuous.

We therefore need a more refined semantics that preserves information about the binding structure of terms, allowing us to delay 'looking up' the value associated with an identifier as long as possible so as to obtain the most relevant/recent version of its definition. This should maintain the essentially call-by-value nature of the calculus, however (we elaborate below on exactly what this means).

We present two reduction strategies with delayed instantiation in §2. The *redex-time* (λ_r) semantics resolves identifiers when in redex position. While this is clean and simple, it is still unnecessarily eager, and so we formulate the *destruct-time* (λ_d) semantics to delay resolving identifiers until their values must be destructed.

Dynamic Rebinding: the λ_{marsh} Calculus With λ_d in place we can consider dynamic rebinding of marshalled values. The key question is this: when a value is moved between scopes, how can the user specify which identifiers should be rebound and which should be fixed? Our answer is embodied in the λ_{marsh} calculus of §3, which contains primitives for packaging a value such that some of its identifiers are fixed to bindings in the current context, while others will be rebound when unpackaged in a new scope (*e.g.*, when the value is moved). Which bindings will be fixed is dynamically determined with respect to a *mark*. Marking is done with an expression form **mark** M **in** e. Here the mark name M is taken from a new syntactic class (not subject to binding); it names the surrounding declaration context. Packaging and unpackaging is done by expressions **marshal** M e and **unmarshal** M e, which are both with respect to a mark. An expression **marshal** M e will first reduce e to a value u, and copy all bindings within the nearest enclosing **mark** M; these bindings are essentially static. Identifiers of u not bound within the mark are recorded in a type environment within the packaged value, which has form **marshalled** Γ u, and can be rebound. For example:

$$\begin{array}{lcl} \textbf{let}\ x_1 = 5\ \textbf{in} & \longrightarrow & \textbf{let}\ x_1 = 5\ \textbf{in} \\ \textbf{mark}\ M\ \textbf{in} & & \textbf{mark}\ M\ \textbf{in} \\ \textbf{let}\ y_1 = 6\ \textbf{in} & & \textbf{let}\ y_1 = 6\ \textbf{in} \\ \textbf{marshal}\ M\ (x_1, y_1) & & \textbf{marshalled}\ \ (x_1{:}\mathsf{int})(\\ & & \quad \textbf{let}\ y_1 = 6\ \textbf{in}\ (x_1, y_1)) \end{array}$$

Because y_1 is defined within the mark M, its definition is copied into the package, while x_1 is defined outside of M, so it is simply noted in the captured type environment. When this package is unmarshalled using **unmarshal** with respect to some mark M', x_1 will be rebound to a definition outside M', subject to a dynamic type environment check.

To indicate more concretely how λ_{marsh} can form the basis for a distributed programming language that supports mobile code, we sketch an extension with concurrency, communication and external library functions, giving examples showing how wrappers for encapsulating untrusted code can be expressed.

Dynamic Update: the λ_{update} Calculus Dynamic updating also requires dynamic rebinding and delayed variable instantiation. We again extend λ_d, here with a simple **update** primitive that allows a program variable to be rebound to a new expression. The resulting λ_{update} calculus is given in §4. As an example, consider the expression on the left below:

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

The **update** expression indicates that an update is possible at the point during evaluation when **update** appears in redex position. At that run-time point the user can supply an update of the form $\{w \Leftarrow e\}$, indicating that w should be rebound to expression e. In the example this update is $\{y \Leftarrow (x_1, 6)\}$; the let-binder for y_1 is modified accordingly yielding the expression on the right above, and thence a final result of 5. Here any identifier in scope at the update point can be rebound, to an expression that may mention identifiers in scope at its binding point. We define what it means for an update to be well-typed with respect to a program; applying well-typed updates preserves typing. The use of λ_d enables us to deal simply and cleanly with higher-order functions, largely ignored in past work. We imagine λ_{update} will form the core of future calculi that include other desirable features, such as state transformation, abstract types, changing the types of variables, multithreading, etc.

2. Call-by-value λ-calculus revisited

This section reconsiders the call-by-value lambda calculus, exploring refined operational semantics that instantiate identifiers at different times. We take a standard syntax:

$$\begin{array}{llcl} \text{Identifiers} & x, y, z & & \\ \text{Integers} & n & & \\ \text{Types} & T & ::= & \mathsf{int} \mid \mathsf{unit} \mid T * T' \mid T \to T' \\ \text{Expressions} & e & ::= & z \mid n \mid () \mid (e, e') \mid \pi_r e \\ & & \mid & \lambda z{:}T.e \mid e e' \mid \textbf{let}\ z = e\ \textbf{in}\ e' \\ & & \mid & \textbf{letrec}\ z = \lambda x{:}T.e\ \textbf{in}\ e' \end{array}$$

Construct-time λ_c

$$\begin{array}{llll}
\text{Values} & v & ::= & n \mid () \mid (v, v') \mid \lambda z{:}T.e \\
\text{Atomic evaluation contexts} & A & ::= & (\_, e) \mid (v, \_) \mid \pi_r \_ \mid \_e \mid v\_ \mid \textbf{let } z = \_ \textbf{ in } e \\
\text{Evaluation contexts} & E & ::= & \_ \mid E.A
\end{array}$$

$$\begin{array}{llll}
\text{(proj)} & \pi_r(v_1, v_2) & \longrightarrow & v_r \\
\text{(app)} & (\lambda z{:}T.e)v & \longrightarrow & \{v/z\}e \\
\text{(let)} & \textbf{let } z = v \textbf{ in } e & \longrightarrow & \{v/z\}e \\
\text{(letrec)} & \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } e' & \longrightarrow & \{\lambda x{:}T.\textbf{letrec } z = \lambda x{:}T.e \textbf{ in } e/z\}e' \quad \text{if } z \neq x
\end{array}
\qquad \frac{e \longrightarrow e'}{E.e \longrightarrow E.e'}$$

Redex-time λ_r and Destruct-time λ_d

Common Syntax and Semantics

$$\begin{array}{llll}
\text{Values} & u & ::= & n \mid () \mid (u, u') \mid \lambda z{:}T.e \mid \textbf{let } z = u \textbf{ in } u' \mid \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } u \\
\text{Atomic evaluation contexts} & A_1 & ::= & (\_, e) \mid (u, \_) \mid \pi_r \_ \mid \_e \mid u\_ \mid \textbf{let } z = \_ \textbf{ in } e \\
\text{Atomic bind contexts} & A_2 & ::= & \textbf{let } z = u \textbf{ in } \_ \mid \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } \_ \\
\text{Evaluation contexts} & E_1 & ::= & \_ \mid E_1.A_1 \\
\text{Bind contexts} & E_2 & ::= & \_ \mid E_2.A_2 \\
\text{Reduction contexts} & E_3 & ::= & \_ \mid E_3.A_1 \mid E_3.A_2
\end{array}
\qquad \frac{e \longrightarrow e'}{E_3.e \longrightarrow E_3.e'}$$

$$\begin{array}{llll}
\text{(proj)} & \pi_r(E_2.(u_1, u_2)) & \longrightarrow & E_2.u_r \\
\text{(app)} & (E_2.(\lambda z{:}T.e))u & \longrightarrow & E_2.\textbf{let } z = u \textbf{ in } e \quad \text{if } \mathrm{fv}(u) \notin \mathrm{hb}(E_2)
\end{array}$$

Redex-time Instantiation Semantics

$$\begin{array}{lllll}
\text{(inst)} & \textbf{let } z = u \textbf{ in } E_3.z & \longrightarrow & \textbf{let } z = u \textbf{ in } E_3.u & \text{if } z \notin \mathrm{hb}(E_3) \text{ and } \mathrm{fv}(u) \notin z, \mathrm{hb}(E_3) \\
\text{(instrec)} & \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_3.z & \longrightarrow & \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_3.\lambda x{:}T.e & \text{if } z \notin \mathrm{hb}(E_3) \text{ and } \mathrm{fv}(\lambda x{:}T.e) \notin \mathrm{hb}(E_3)
\end{array}$$

Destruct-time Syntax Extension and Instantiation Semantics

$$\begin{array}{l}
\text{Values} \\
\quad u \quad ::= \quad ... \mid z \\[1em]
\text{Destruct contexts} \\
\quad R \quad ::= \quad \pi_r \_ \mid \_u
\end{array}
\qquad
\begin{array}{llll}
\text{(inst-1)} & \textbf{let } z = u \textbf{ in } E_3.R.E_2.z & \longrightarrow & \textbf{let } z = u \textbf{ in } E_3.R.E_2.u \\
& \quad \text{if } z \notin \mathrm{hb}(E_3, E_2) \text{ and } \mathrm{fv}(u) \notin z, \mathrm{hb}(E_3, E_2) & & \\
\text{(inst-2)} & R.E_2.\textbf{let } z = u \textbf{ in } E_2'.z & \longrightarrow & R.E_2.\textbf{let } z = u \textbf{ in } E_2'.u \\
& \quad \text{if } z \notin \mathrm{hb}(E_2') \text{ and } \mathrm{fv}(u) \notin z, \mathrm{hb}(E_2') & & \\
\text{(instrec-1)} & \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_3.R.E_2.z & \longrightarrow & \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_3.R.E_2.\lambda x{:}T.e \\
& \quad \text{if } z \notin \mathrm{hb}(E_3, E_2) \text{ and } \mathrm{fv}(\lambda x{:}T.e) \notin \mathrm{hb}(E_3, E_2) & & \\
\text{(instrec-2)} & R.E_2.\textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_2'.z & \longrightarrow & R.E_2.\textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_2'.\lambda x{:}T.e \\
& \quad \text{if } z \notin \mathrm{hb}(E_2') \text{ and } \mathrm{fv}(\lambda x{:}T.e) \notin \mathrm{hb}(E_2') & &
\end{array}$$

Figure 1: Three Call-by-Value Lambda Calculi

where r ranges over $\{1, 2\}$. Expressions are taken up to alpha equivalence (though contexts are not). It is simply-typed, with a typing judgement $\Gamma \vdash e{:}T$ defined as usual, where Γ ranges over sequences of $z{:}T$ pairs. We omit the typing rules for brevity.

2.1 Construct-time

The standard semantics, here called the *construct-time* semantics, is recalled at the top of Fig. 1. We define a small-step reduction relation $e \longrightarrow e'$, using evaluation contexts E, and a run-time-error predicate e err (the rules for the latter are elided, but as usual, projections from non-pairs and application to non-functions are the only conditions giving rise to errors). Context composition and application are both written with a dot, *e.g.*, $E.E'$ and $E.e$, instead of the usual heavier brackets $E[e]$. Standard capture-avoiding substitution of e for z in e' is written $\{e/z\}e'$. We write $\mathrm{hb}(E)$, defined below, for the list of binders around the hole of E. For now we will be concerned only with the behaviour of closed expressions, without external library functions. The choice of a small-step semantics will be important when we add dynamic rebinding and communication later.

2.2 Redex-time

The redex-time and destruct-time semantics are shown in Fig. 1, with common syntax and semantics presented first. Instead of substituting bindings of identifiers to values, as in the construct-time (app) and (let), both semantics introduce a **let** to record a binding of the abstraction's formal parameter to the application argument, *e.g.*,

$$(\lambda z{:}T.e)u \quad \longrightarrow \quad \textbf{let } z = u \textbf{ in } e$$

This is reminiscent of an explicit substitution [ACCL90], save that here the **let** will not be percolated through the term structure, and also of the λ_{let}-calculus [AFM$^+$95], though we are in a CBV not CBN setting, and do not allow commutation of **let**s. In contrast, we must preserve let-binding structure, since our later rebinding and update primitives will depend on it.

Example (1) in Fig. 2 illustrates (app), contrasting it with the substitution approach of the construct-time semantics. Note that the resulting **let** $z = 8$ **in** 7 is a λ_r (and λ_d) value. Because values may involve **let**s, some clean-up is needed to extract the usual final

	Construct-time λ_c	**Redex-time λ_r**	**Destruct-time λ_d**
(1)	$(\lambda z.7)8$	$(\lambda z.7)8$	$(\lambda z.7)8$
$\longrightarrow$	7	$\textbf{let } z = 8 \textbf{ in } 7$	$\textbf{let } z = 8 \textbf{ in } 7$
(2)	$\textbf{let } x = 5 \textbf{ in } \pi_1(x, x)$	$\textbf{let } x = 5 \textbf{ in } \pi_1(x, x)$	$\textbf{let } x = 5 \textbf{ in } \pi_1(x, x)$
$\longrightarrow$	$\pi_1(5, 5)$	$\textbf{let } x = 5 \textbf{ in } \pi_1(5, x)$	$\textbf{let } x = 5 \textbf{ in } x$
$\longrightarrow$	5	$\textbf{let } x = 5 \textbf{ in } \pi_1(5, 5)$	
$\longrightarrow$		$\textbf{let } x = 5 \textbf{ in } 5$	
(3)	$\textbf{let } x = (5, 6) \textbf{ in let } y = x \textbf{ in } \pi_1 y$	$\textbf{let } x = (5, 6) \textbf{ in let } y = x \textbf{ in } \pi_1 y$	$\textbf{let } x = (5, 6) \textbf{ in let } y = x \textbf{ in } \pi_1 y$
$\longrightarrow$	$\textbf{let } y = (5, 6) \textbf{ in } \pi_1 y$	$\textbf{let } x = (5, 6) \textbf{ in let } y = (5, 6) \textbf{ in } \pi_1 y$	$\textbf{let } x = (5, 6) \textbf{ in let } y = x \textbf{ in } \pi_1 x$
$\longrightarrow$	$\pi_1(5, 6)$	$\textbf{let } x = (5, 6) \textbf{ in let } y = (5, 6) \textbf{ in } \pi_1(5, 6)$	$\textbf{let } x = (5, 6) \textbf{ in let } y = x \textbf{ in } \pi_1(5, 6)$
$\longrightarrow$	5	$\textbf{let } x = (5, 6) \textbf{ in let } y = (5, 6) \textbf{ in } 5$	$\textbf{let } x = (5, 6) \textbf{ in let } y = x \textbf{ in } 5$

Figure 2: Call-by-Value Lambda Calculi Examples

result, for which we define

$$\begin{array}{rcl}
[\![\, n \,]\!] & = & n \\
[\![\, () \,]\!] & = & () \\
[\![\, (u, u') \,]\!] & = & ([\![\, u \,]\!], [\![\, u' \,]\!]) \\
[\![\, \lambda x{:}T.e \,]\!] & = & \lambda x{:}T.e \\
[\![\, \textbf{let } z = u \textbf{ in } u' \,]\!] & = & \{[\![\, u \,]\!]/z\}[\![\, u' \,]\!] \\
\multicolumn{3}{l}{[\![\, \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } u \,]\!]} \\
\multicolumn{3}{l}{= \{\lambda x{:}T.\textbf{letrec } z = \lambda x{:}T.e \textbf{ in } e/z\}[\![\, u \,]\!] \quad \text{if } z \neq x} \\
[\![\, z \,]\!] & = & z
\end{array}$$

taking any value (λ_r or λ_d) and substituting out the **let**s.

The semantics must allow reduction under **let**s – in addition to the atomic evaluation contexts A we had above (here A_1) we now have the binding contexts $A_2 ::= \textbf{let } z = u \textbf{ in } \_$. Reduction is closed under both. Redex-time variable resolution is handled with the (inst) rule, which resolves an occurrence of the identifier z in redex position with the innermost enclosing **let** that binds that identifier. The side-condition $z \notin \text{hb}(E_3)$ ensures that the correct binding of z is used. Here $\text{hb}(E)$ denotes the list of identifiers that bind around the hole of a context E, is defined by $\text{hb}(\_) = []$; $\text{hb}(E.(\textbf{let } z = e \textbf{ in } \_)) = \text{hb}(E), z$; $\text{hb}(E.(\textbf{letrec } z = \lambda x{:}T.e \textbf{ in } \_)) = \text{hb}(E), z$; and $\text{hb}(E.A) = \text{hb}(E)$ for any other atomic context A. We overload $\in$ for lists. The other side-condition, $\text{fv}(u) \notin z, \text{hb}(E_3)$, which can always be achieved by alpha conversion, prevents identifier capture, making E_3 and $\textbf{let } z = u \textbf{ in } \_$ transparent for u. Here $\text{fv}(\_)$ denotes the set of free identifiers of an expression or context.

Example (2) in Fig. 2 illustrates identifier instantiation. While the construct-time strategy substitutes for x immediately, the redex-time strategy instantiates x under the **let**, following the evaluation order. Both this and the first example also illustrate a further aspect of the redex-time calculus: values u include let-bindings of the form $\textbf{let } z = u \textbf{ in } u'$. Intuitively, this is because a value should 'carry its bindings with it' preventing otherwise stuck applications, *e.g.*, $(\lambda x{:}\text{int}.x)(\textbf{let } z = 3 \textbf{ in } 5)$ or (for an example where the **let** is not garbage) $(\lambda f{:}(\text{int} \to \text{int}).x\ 2)(\textbf{let } z = 3 \textbf{ in } \lambda x{:}\text{int}.z)$. Note that identifers are not values, so z, (z, z) and $\textbf{let } z = 3 \textbf{ in } (z, z)$ are not values. Values may contain free identifiers under lambdas, as usual, so $\lambda x{:}\text{int}.z$ is an open value and $\textbf{let } z = 3 \textbf{ in } \lambda x{:}\text{int}.z$ is a closed value.

The (proj) and (app) rules are straightforward except for the additional binding context E_2. This is necessary as a value may now have some let bindings around a pair or lambda; terms such as $\pi_1(\textbf{let } z = 3 \textbf{ in } (4, 5))$ or (more interestingly) $\pi_1(\textbf{let } z = 3 \textbf{ in } (\lambda x{:}\text{int}.z, 5))$ would otherwise be stuck. The side condition for (app) can always be achieved by alpha conversion; it prevents capture.

2.3 Destruct-time

The redex-time strategy is appealingly simple, but it instantiates earlier than necessary. In example (2) in Fig. 2, both occurrences of x are instantiated before the projection reduction. However, we could delay resolving x until *after* the projection; we see this behaviour in the destruct-time semantics in the third column. In many dynamic rebinding scenarios it is desirable to instantiate as late as possible.[1] For example, in repeatedly-mobile code, we want to instantiate each identifier only as needed to always pick up local definitions. Similarly, for dynamically updateable code we want to delay looking up a variable as long as possible, so as to acquire the most recent version.

To instantiate as late as possible, while remaining call-by-value, we instantiate only identifiers that are immediately under a projection or on the left-hand-side of an application. In these 'destruct' positions their values are about to be deconstructed, and so their outermost pair or lambda structure must be made manifest. The *destruct contexts* $R ::= \pi_r\_ \mid \_u$ can be seen as the outer parts of the construct-time (proj) and (app) redexes. The choice of destruct contexts is determined by the basic redexes – for example, if we added arithmetic operations, we would need to instantiate identifiers of int type before using them.

The essential change from the redex-time semantics is that now any identifier is a value ($u ::= ... \mid z$). The (proj) and (app) rules are unchanged. The (inst) rule is replaced by two that together instantiate identifiers in destruct contexts R. The first (inst-1) copes with identifiers that are let-bound outside a destruct context, *e.g.*:

$$\textbf{let } z = (1, 2) \textbf{ in } \pi_1 z \quad \longrightarrow \quad \textbf{let } z = (1, 2) \textbf{ in } \pi_1(1, 2)$$

whereas in (inst-2) the let-binder and destruct context are the other way around:

$$\pi_1(\textbf{let } z = (1, 2) \textbf{ in } z) \quad \longrightarrow \quad \pi_1(\textbf{let } z = (1, 2) \textbf{ in } (1, 2))$$

Further, we must be able to instantiate under nested bindings between the binding in question and its use. Therefore, (inst-2) must allow additional bindings E_2 and E_2' between R and the **let** and

[1]"It is the conventional wisdom of distributed programming that in any cases of this sort early binding is extremely wicked, and every opportunity must be taken to allow for variability." [Nee93].

between the **let** and z. Similarly, (inst-1) must allow bindings E_2 between the R and z; it must allow both binding and evaluation contexts E_3 between the **let** and the R, *e.g.*, for the instance

$$\begin{array}{ll} & \textbf{let } z = (1,(2,3)) \textbf{ in } \pi_1(\pi_2 z) \\ \longrightarrow & \textbf{let } z = (1,(2,3)) \textbf{ in } \pi_1(\pi_2(1,(2,3))) \end{array}$$

with $E_3 = \pi_1\_$, $R = \pi_2\_$ and $E_2 = \_$. The conditions $z \notin \mathrm{hb}(E_3, E_2)$ and $z \notin \mathrm{hb}(E_2')$ ensure that the correct binding of z is used; the other conditions prevent capture and can always be achieved by alpha equivalence.

Example (3) illustrates a chain of instantiations, from outside in for λ_r and from inside out for λ_d.

2.4 Properties

This subsection gives properties of our various λ-calculi: sanity checks to confirm that our definitions are coherent and more substantial results showing that λ_r and λ_d are essentially CBV. Details of proofs can be found in the technical report [BHS$^+$03].

First, we recall the important unique decomposition property of evaluation contexts for λ_c, essentially as in [FF87, p. 200], and generalise it to the more subtle evaluation contexts of λ_r and λ_d:

THEOREM 1 (UNIQUE DECOMPOSITION FOR λ_r AND λ_d). *Let e be a closed expression. Then, in both the redex-time and destruct-time calculi, exactly one of the following holds: (1) e is a value; (2) e err; (3) there exists a triple (E_3, e', rn) such that $E_3.e' = e$ and e' is an instance of the left-hand side of rule rn. Furthermore, if such a triple exists then it is unique.*

(Note that the destruct-time error rules defining e err, which have been elided, must include cases for identifiers in destruct contexts that are not bound by enclosing **let**s and so are not instantiatable, giving stuck non-value expressions.) Determinacy is a trivial corollary. We also have type preservation and type safety properties for the three calculi.

THEOREM 2 (TYPE PRESERVATION FOR λ_c, λ_r AND λ_d). *If $\Gamma \vdash e{:}T$ and $e \longrightarrow e'$ then $\Gamma \vdash e'{:}T$.*

THEOREM 3 (SAFETY FOR λ_c, λ_r AND λ_d). *If $\vdash e{:}T$ then $\neg(e$ err$)$.*

Finally we show that all three calculi are observationally equivalent, hence that both λ_r and λ_d are essentially call-by-value. As we noted earlier, values in λ_r and λ_d may need to be 'cleaned-up' to exactly correspond to λ_c values. The proof of this is non-trivial; it involves constructing a tight correspondence between reduction steps in the three calculi.

THEOREM 4 (OBSERVATIONAL EQUIVALENCE).

1. *If $\vdash e{:}\mathsf{int}$ and $e \longrightarrow_c^* n$ then $e \longrightarrow_r^* u$ and $e \longrightarrow_d^* u'$ for some u and u' with $[\![u]\!] = [\![u']\!] = n$.*
2. *If $\vdash e{:}\mathsf{int}$ and $e \longrightarrow_r^* u$ (or $e \longrightarrow_d^* u$) then for some n we have $e \longrightarrow_c^* n$ and $[\![u]\!] = n$.*

Proof Sketch The proof technique is the same for both claims: generalise the claim to arbitary type and proceed to construct a bisimulation (modulo possible instantiations of letrec bindings) that captures a tight operational correspondence between reductions in the different calculi. To do so, we introduce intermediate caluli with annotated lets, distinguishing lets that, in the λ_c reduction sequence, correspond to substitutions from those that have yet to be reached. Additional transitions move value-lets from the latter to the former. Bisimulations can then be constructed by factoring simulations through these intermediate calculi. A key notion in the simulation proofs is that of instantiation normal form. Essentially a term is in instantiation normal form if it can not do an instantiation reduction. It is important that this form is always finitely reachable by reduction from any term. Finally, we use the bisimulation and some auxilary lemmas to prove the generalised claim (the claim as stated in the theorem avoids the complication of possible letrec unfoldings).

3. A Dynamic Rebinding Calculus: λ_{marsh}

Many applications require a mix of dynamically and statically bound variables. Consider sending a function value between machines. It might contain identifiers for

(1) ubiquitous standard library calls, *e.g.*, $print$, which should be rebound at the destination;
(2) application-specific location-dependent library calls, *e.g.*, routing functions, which should be rebound at the destination;
(3) application code which is not location-dependent but (for performance) should be rebound rather than sent; and
(4) other let-bound application values, which should be sent with it.

Moreover, for both (1) and (2) one may wish the rebinding to be to non-standard definitions, to securely encapsulate (sandbox) untrusted code.

In this section we develop a calculi to support all of the above. The calculus λ_{marsh} extends the destruct-time λ_d-calculus of §2.3 with high-level representations of *marshalled* values and primitives to manipulate them. We make two main choices. First, to have as intuitive a semantics as possible we want dynamic rebinding to only occur when unmarshalling values, not during normal computation. Second, to allow the programmer to cleanly and flexibly notate which definitions should be fixed and which should be rebindable, we introduce *marks* **mark** M **in** e which name contexts. Marshal and unmarshal operations **marshal** M e and **unmarshal** M e are each with respect to a mark: a **marshal** M u packages the value u together with all the bindings within the closest enclosing **mark** M (thus fixing them); it cuts any bindings of identifiers in u that cross that **mark** M (thus making them rebindable). When the packaged value is unpackaged by an **unmarshal** M' _, the latter identifiers are rebound to binders outside the closest enclosing **mark** M'.

The **mark** M **in** e construct does *not* bind M; marks have global meaning across a distributed system. Allowing the choice of context to be made differently for each **marshal** and **unmarshal** provides important flexibility, especially for implementing secure encapsulation; note that we have just a single class of identifiers, rather than dynamic and static forms. In the simplest practical case each program might have a single **mark** Lib **in** _, distinguishing library code, defined above the mark, from application code, defined below it.

For simplicity, λ_{marsh} simulates communication using beta-reduction (in fact, λ_d (inst) reduction), and omits treatment of (1), focusing on the more interesting cases of rebinding application-specific libraries. At the end of this section we sketch $\lambda_{\mathrm{marsh}}^{\mathrm{io}}$, which straightforwardly extends λ_{marsh} with communication and external identifiers, and discuss alternative design choices.

3.1 Syntax

The λ_{marsh} syntax and an example, discussed below, are given in Fig. 3; the new semantic rules are given in Fig. 4 (error rules omitted). The calculus requires a more elaborate treatment of alpha

Syntax

Integers n Identifiers x, y, z Tags i, j, k Context marks M

Type environments	Γ		finite partial functions from (identifier,tag) pairs to types
Types	T	$::=$	$\mathsf{int} \mid \mathsf{unit} \mid T * T' \mid T \to T' \mid \mathsf{Marsh}\ T$
Expressions	e	$::=$	$z_i \mid n \mid () \mid (e, e') \mid \pi_r e \mid \lambda x_i{:}T.e \mid e e' \mid \mathbf{let}\ z_k{:}T = e\ \mathbf{in}\ e' \mid \mathbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \mathbf{in}\ e' \mid$ $\mathbf{mark}\ M\ \mathbf{in}\ e \mid \mathbf{marshal}\ M\ e \mid \mathbf{marshalled}\ \Gamma\ u \mid \mathbf{unmarshal}\ M\ e$

Example

$$\begin{array}{l}
\mathbf{let}\ y_1{:}\mathsf{int} = 6\ \mathbf{in}\\
\mathbf{mark}\ M\ \mathbf{in}\\
\mathbf{let}\ x_1{:}\mathsf{Marsh}\ (\mathsf{int} * \mathsf{int}) = (\\
\quad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
\quad \boxed{\mathbf{marshal}\ M\ (y_1, z_1))}\ \mathbf{in}\\
\mathbf{let}\ y_2{:}\mathsf{int} = 7\ \mathbf{in}\\
\mathbf{mark}\ M'\ \mathbf{in}\\
\mathbf{unmarshal}\ M'\ x_1
\end{array}
\xrightarrow{(\text{marshal})}
\begin{array}{l}
\mathbf{let}\ y_1{:}\mathsf{int} = 6\ \mathbf{in}\\
\mathbf{mark}\ M\ \mathbf{in}\\
\boxed{\mathbf{let}\ x_1{:}T = (}\\
\quad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
\quad \mathbf{marshalled}\ (y_0{:}\mathsf{int})\ (\\
\qquad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
\qquad (y_0, z_1)))\ \mathbf{in}\\
\mathbf{let}\ y_2{:}\mathsf{int} = 7\ \mathbf{in}\\
\mathbf{mark}\ M'\ \mathbf{in}\\
\mathbf{unmarshal} M'\ \boxed{x_1}
\end{array}
\xrightarrow{(\text{inst-1})}
\begin{array}{l}
\mathbf{let}\ y_1{:}\mathsf{int} = 6\ \mathbf{in}\\
\mathbf{mark}\ M\ \mathbf{in}\\
\mathbf{let}\ x_1{:}T = (\\
\quad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
\quad \mathbf{marshalled}\ (y_0{:}\mathsf{int})\ (\\
\qquad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
\qquad (y_0, z_1)))\ \mathbf{in}\\
\mathbf{let}\ y_2{:}\mathsf{int} = 7\ \mathbf{in}\\
\mathbf{mark}\ M'\ \mathbf{in}\\
\boxed{\mathbf{unmarshal}\ M'}\ (\\
\quad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
\quad \boxed{\begin{array}{l}\mathbf{marshalled}\ (y_0{:}\mathsf{int})\ (\\ \quad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\ \quad (y_0, z_1)))\end{array}}
\end{array}
\xrightarrow{(\text{unmarshal})}
\begin{array}{l}
\mathbf{let}\ y_1{:}\mathsf{int} = 6\ \mathbf{in}\\
\mathbf{mark}\ M\ \mathbf{in}\\
\mathbf{let}\ x_1{:}T = (\\
\quad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
\quad \mathbf{marshalled}\ (y_0{:}\mathsf{int})\ (\\
\qquad \mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
\qquad (y_0, z_1)))\ \mathbf{in}\\
\mathbf{let}\ y_2{:}\mathsf{int} = 7\ \mathbf{in}\\
\mathbf{mark}\ M'\ \mathbf{in}\\
\mathbf{let}\ z_1{:}\mathsf{int} = 3\ \mathbf{in}\\
(y_2, z_1)
\end{array}$$

where $T = \mathsf{Marsh}\ (\mathsf{int} * \mathsf{int})$

Figure 3: Dynamic Rebinding Calculus λ_{marsh}: Syntax and Example

equivalence than λ_d. There – as usual for λ-calculi – we had to use alpha equivalence during normal computation steps, to avoid mistaken capture of identifiers as the rules move subterms between different scopes. Here that is still required, but occurrences of the 'same' identifier under different bindings must be related so that the identifier can be marshalled with respect to one and unmarshalled with respect to another. Accordingly, instead of working with identifiers x, we work with *variables* x_i that are pairs of an identifier x and a *tag* i, similar to the external and internal names used in some module systems. Alpha equivalence changes only the tags; tags for different identifiers lie in different namespaces, so *e.g.*,

$$\lambda x_1{:}T.x_1 = \lambda x_2{:}T.x_2 \neq \lambda y_2{:}T.y_2 \qquad \text{and}$$
$$\lambda x_1{:}T.\lambda y_1{:}T.(x_1, y_1) = \lambda x_2{:}T.\lambda y_3{:}T.(x_2, y_3)$$

In practice tags would not appear in source programs; they are needed only for the semantics. The fv(_) and hb(_) functions now give sets and lists of variables, respectively, not identifiers.

3.2 Example

As an example, consider the expression on the left of Fig. 3. The value (y_1, z_1) is marshalled with respect to the context marked M, where $y = 6$, but unmarshalled with respect to the context M', where $y = 7$. The z_1, on the other hand, is bound *below* mark M, so its binding $z_1 = 3$ is grabbed and carried with it.

The reduction sequence is shown in the Figure, boxing key parts of $\boxed{redexes}$ and *contracta*. The first reduction step copies the bindings that are inside **mark** M and around the **marshal** expression (here just $z_1 = 3$), ensuring that these have static-binding semantics. This gives a value

$$\mathbf{marshalled}\ (y_0{:}\mathsf{int})\ (\mathbf{let}\ z_1 = 3\ \mathbf{in}\ (y_0, z_1))$$

This **marshalled** Γ u form would not occur in source programs. The free variables of u are subject to rebinding when this is unmarshalled, so we regard all of fv(u) as bound by Γ in **marshalled** Γ u. This is emphasised in the example by showing a y_0 alpha-variant.

The second step instantiates the x_1 under the (**unmarshal** M' _) with its value **let** $z_1 = 3$ **in** ...**marshalled**.... (In this case the outer z_1 **let** is redundant but in more complex cases it would not be, *e.g.*, if x_1 were bound to a pair of the marshalled value and some other value mentioning z_1.)

The third step performs the unmarshal, rebinding the y_0 in the packaged value **let** $z_1 = 3$ **in** (y_0, z_1) to the innermost y_i binder outside **mark** M' – here, to y_2. It also discards the now-redundant bindings.

Modulo final instantiation, the result is $(7, 3)$ not $(6, 3)$, showing the y_1 and z_1 have been treated dynamically and statically respectively. For contrast, putting the first **let** $y_1 = 6$ inside the first mark M would give $(6, 3)$.

3.3 Semantics

Turning now to the details of the rules, the (proj), (app) and (inst-r) rules are as in λ_d but with z_k instead of z. In the (marshal) and (unmarshal) rules we abuse notation, writing the context **mark** M **in** _ as **mark** M. The (marshal) rule copies all bindings and marks between the **marshal** M _ and the closest enclosing **mark** M, using the bindmark(_) auxiliary to extract the bind and mark components of a context E_3, discarding the evaluation context components: bindmark(_) = _, bindmark($E_3.A_1$) = bindmark(E_3), and bindmark($E_3.A_2$) = bindmark(E_3).A_2. The predicate dhb(E_3) holds iff the hole-binders of E_3 are all distinct (which can always be made so by alpha conversion). The auxiliary env(E_3) extracts the type environment of the hole-binders of E_3, so they can be recorded in the **marshalled** value.

The (unmarshal) rule rebinds the fv(u) to the let-binders in E_3 around the nearest enclosing **mark** M, using the auxiliary func-

Values	u	$::=$	$n \mid () \mid (u, u') \mid \lambda x_i{:}T.e \mid$ **let** $z_k{:}T = u$ **in** $u' \mid$ **letrec** $z_k{:}T' = \lambda x_i{:}T.e$ **in** $u \mid z_i$
		$\mid$	**mark** M **in** $u \mid$ **marshalled** $\Gamma\ u$
Atomic evaluation contexts	A_1	$::=$	$(\_, e) \mid (u, \_) \mid \pi_r\_ \mid \_e \mid (\lambda x_i{:}T.e)\_ \mid$ **let** $z_k{:}T = \_$ **in** e
		$\mid$	**marshal** $M\ \_ \mid$ **unmarshal** $M\ \_$
Atomic bind and mark contexts	A_2	$::=$	**let** $z_k{:}T = u$ **in** $\_ \mid$ **letrec** $z_k{:}T' = \lambda x_i{:}T.e$ **in** $\_$
		$\mid$	**mark** M **in** $\_$
Evaluation contexts	E_1	$::=$	$\_ \mid E_1.A_1$
Bind and mark contexts	E_2	$::=$	$\_ \mid E_2.A_2$
Reduction contexts	E_3	$::=$	$\_ \mid E_3.A_1 \mid E_3.A_2$
Destruct contexts	R	$::=$	$\pi_r\_ \mid \_u \mid$ **unmarshal** $M\ \_$

Rules (proj), (app), (inst-r), (instrec-r) are exactly as in λ_d except for z_k replacing z and the addition of explicit types. These reductions are closed under E_3, whereas the (marshal) and (unmarshal) rules are global.

(marshal) $E_3.$**mark** $M.E_3'.$**marshal** $M\ u \longrightarrow E_3.$**mark** $M.E_3'.$**marshalled** $(\text{env}(E_3))\ (\text{bindmark}(E_3').u)$
if $\text{dhb}(E_3)$ and no **mark** M around $\_$ in E_3'

(unmarshal) $E_3.$**mark** $M.E_3'.$**unmarshal**$M.E_2.$**marshalled** $\Gamma\ u \longrightarrow E_3.$**mark** $M.E_3'.S(u)$
if $\text{dhb}(E_3)$, $\text{dhb}(E_3', \text{hb}(E_3))$, $S = \text{rebind}(\Gamma, \text{thb}(E_3))$ is defined, and no **mark** M around $\_$ in E_3'.

Figure 4: Dynamic Rebinding Calculus λ_{marsh}: Semantics

tion rebind$(_,_)$ to construct the appropriate substitution. Here $\text{dhb}(E_3', \text{hb}(E_3))$ holds iff the hole-binders of E_3' are distinct from each other and from all the variables in $\text{hb}(E_3)$ (always possible by alpha conversion). The $\text{thb}(E_3)$ gives the list of (variable,type) pairs, which are the *typed* hole-binders of E_3 (type annotations were added to **let**s to facilitate this). Finally, $\text{rebind}(\Gamma, L)$, for a type environment Γ and list of typed hole-binders L, is a substitution taking each x_i in $\text{dom}(\Gamma)$ to the rightmost x_j in L, if the types correspond appropriately. It is defined by

$$\text{rebind}(\Gamma, [])$$
$$\begin{cases} \text{undefined} & \text{if } \Gamma \text{ nonempty} \\ = \{\} & \text{otherwise} \end{cases}$$

$$\text{rebind}(\Gamma, (L, (x_i{:}T)))$$
$$\begin{cases} \text{undefined, if } \exists j, T'.(x_j{:}T') \in \Gamma \wedge T' \neq T \\ = \{x_i/x_J\} \cup \text{rebind}(\Gamma - x_J, L), \quad \text{otherwise} \\ \quad \text{where } x_J = \{x_j \mid (x_j{:}T) \in \Gamma\} \end{cases}$$

(abusing notation to treat the partial function Γ as a set of tuples and writing $\{x_i/x_J\}$ for the substitution of x_i for all the $x_j \in x_J$). To keep a unique decomposition property the (unmarshal) rule is global, not closed under additional E_3. We briefly justify why the (unmarshal) rule discards its E_2 context: observe the right hand side of the rule and notice that the binders in the E_2 context can no longer be referenced after unmarshalling, the only possible references to the enclosing E_2 are the free variables of u, but subsequent to this reduction these variables are rebound to binders in E_3.

Reduction must take place under a **mark** so A_2 now contains **mark** M **in** $\_$. To maintain a CBV semantics both **marshal** and **unmarshal** should fully reduce their arguments, so they are included in the evaluation contexts A_1. The (unmarshal) rule can only fire if the argument to **unmarshal** is of the form **marshalled** $\Gamma\ u$, so the destruct contexts must include **unmarshal** $M\ \_$.

There are several choices embodied in the semantics. First, in (marshal) $\text{bindmark}(E_3')$ records the marks of E_3' as well as its let-bindings, so that uses of **marshal** and **unmarshal** within u will behave as expected. Second, in (marshal) we record the full type environment $\text{env}(E_3)$, not just its restriction to $\text{fv}(u)$. The latter would be more liberal (more unmarshals would succeed) but we believe would lead to code that is hard to maintain: success of an unmarshal would depend on the free variables of the marshalled value, instead of simply on the binders above the mark used for marshalling. Third, if there is shadowing of identifiers outside a mark then a **marshalled** $\Gamma\ u$ may have Γ with $x_i{:}T$ and $x_j{:}T'$ for $T \neq T'$, in which case (unmarshal) will always fail. One could check this at (marshal)-time, or indeed forbid shadowing outside marks.

3.4 Typing and Run-Time Errors

In some cases one would expect dynamic rebinding to require a run-time check to ensure safety, *e.g.*, if code is sent to a site that may or may not provide some resource it requires. For λ_{marsh} we have new run-time errors, if a **marshal** or an **unmarshal** refers to a mark which is not in scope, or if at (unmarshal)-time the environment does not have the required binders at the correct types. At the very least, however, one would like a type system to exclude all run-time errors except these. This can be done by a simple type system, as usual but with a type $\mathsf{Marsh}\ T$ of marshalled type-T values, and rules

$$\frac{\Gamma \vdash e{:}T}{\Gamma \vdash \textbf{mark}\ M\ \textbf{in}\ e{:}T} \qquad \frac{\Gamma \vdash e{:}T}{\Gamma \vdash \textbf{marshal}\ M\ e{:}\mathsf{Marsh}\ T}$$

$$\frac{\Gamma \vdash e{:}\mathsf{Marsh}\ T}{\Gamma \vdash \textbf{unmarshal}\ M\ e{:}T} \qquad \frac{\Gamma' \vdash u{:}T}{\Gamma \vdash \textbf{marshalled}\ \Gamma'\ u{:}\mathsf{Marsh}\ T}$$

Partitioning the run-time errors into e err for the usual projection/application errors, together with unmarshalling of values not of the form **marshalled** $\Gamma\ u$, and e err' for the new errors above (their rules are elided), we have:

THEOREM 5 (UNIQUE REDEX/CONTEXT DECOMPOSITION). *Let e be a closed λ_{marsh} expression. Then exactly one of the following holds: (1) e is a value; (2) e err; (3) e err'; (4) there exist E_3, e_0, rn such that $E_3.e_0 = e$ and e_0 is an instance of the left-hand side of rule $rn \in$* (proj,app,inst-r,instrec-r). *(5) there exists rn $\in$*(marshal),(unmarshal) *such that e is an instance of the left-hand side of rule rn. Furthermore, if such a triple or rn exists then it is unique.*

THEOREM 6 (TYPE PRESERVATION FOR λ_{marsh}).

Simple: $P =$

$$\left(\begin{array}{l} t_1{:}\textbf{let}\ here_0 = \text{“site 1”}\ \textbf{in}\\ \quad\textbf{mark}\ AppLib\ \textbf{in}\\ \quad\textbf{let}\ \_ = print_0\, here_0\ \textbf{in}\\ \quad c!\textbf{marshal}\ AppLib\ (\lambda x_0{:}\mathsf{unit}.print_0\, here_0)\end{array}\right) \Bigg| \left(\begin{array}{l} t_2{:}\textbf{let}\ here_0 = \text{“site 2”}\ \textbf{in}\\ \quad\textbf{mark}\ AppLib\ \textbf{in}\\ \quad c?(\lambda f_0{:}\mathsf{Marsh}\ (\mathsf{unit}\to\mathsf{unit}).(\textbf{unmarshal}\ AppLib\ f_0)())\\ \end{array}\right)$$

Secure encapsulation: $Q =$

$$\left(\begin{array}{l} t_1{:}\textbf{let}\ here_0 = \text{“site 1”}\ \textbf{in}\\ \quad\textbf{mark}\ AppLib\ \textbf{in}\\ \quad\textbf{let}\ \_ = print_0\, here_0\ \textbf{in}\\ \quad c!\textbf{marshal}\ AppLib\ (\lambda x_0{:}\mathsf{unit}.print_0\, here_0)\end{array}\right) \Bigg| \left(\begin{array}{l} t_2{:}\textbf{let}\ here_0 = \text{“site 2”}\ \textbf{in}\\ \quad\textbf{mark}\ TrustedAppLib\ \textbf{in}\\ \quad\textbf{let}\ print_3 = (\lambda s_0{:}\mathsf{string}.\\ \qquad\textbf{let}\ \_ = print_0\text{“sandboxed: ”}\ \textbf{in}\ print_0\, s_0)\ \textbf{in}\\ \quad\textbf{let}\ here_3 = \text{“site 33”}\ \textbf{in}\\ \quad\textbf{mark}\ UntrustedAppLib\ \textbf{in}\\ \quad c?(\lambda f_0{:}\mathsf{Marsh}\ (\mathsf{unit}\to\mathsf{unit}).\\ \qquad\textbf{let}\ g_0 = (\textbf{if}\ trusted()\ \textbf{then}\ \textbf{unmarshal}\ TrustedAppLib\ f_0\\ \qquad\textbf{else}\ \textbf{unmarshal}\ UntrustedAppLib\ f_0)\ \textbf{in}\ g_0())\end{array}\right)$$

Figure 5: Dynamic Rebinding with IO and Communication: $\lambda^{\text{io}}_{\text{marsh}}$ Examples

If $\vdash e{:}T$ *and* $e \longrightarrow e'$ *then* $\vdash e'{:}T$

THEOREM 7 (PARTIAL SAFETY FOR λ_{marsh}).
If $\vdash e{:}T$ *then* $\neg(e\ \mathsf{err})$.

A full language would raise catchable exceptions in the $e\ \mathsf{err}'$ cases, thereby allowing code to dynamically check the presence of resources.

Ideally, of course, one would like a type system that could statically prevent *all* run-time errors, in the case where all parts of the (distributed) system can be type-checked coherently. Unfortunately static typing and dynamic rebinding seem to be at odds. Any sound type system for λ_{marsh} must constrain the contexts around marks, ensuring that when unmarshalling a marshalled value the context of the unmarshal mark contains bindings for all identifiers that were in the context of the marshal mark. The problem is that reduction moves subterms, in particular subterms containing marks, so the shape of the context around a mark can change dynamically. One can devise rather draconian systems that prevent some run-time errors, but it is hard to see what a really useful system could be like. Moreover, in the wide-area setting it is generally impossible to guarantee that all parts are type-checked together, so we believe that the limited guarantees of the simple type system above may have to suffice.

In practice one would expect programs to contain only a few marks. For ML-like languages with second-class module systems it may be desirable to allow marks only between module declarations – a considerable simplification.

3.5 Implementation

The reduction semantics as presented is not proposed as a realistic implementation strategy. Instead of representing bindings by nested **let** terms, and preserving binding scopes in the instantiation rules by copying and α-conversion, we propose to use linked environment frames with sharing, as is done to implement function closures. A function closure consists of the binding variable name, function body, and a pointer to the enclosing environment. The environment consists of frames, each containing a variable name, value, and a link pointer to the parent frame. For λ_d, variables as well as functions are values; therefore we introduce *variable closures*, consisting of a variable name and an environment pointer through which to look it up. Only when the variable closure appears in a destruct context is the pointer followed to obtain its value. For λ_{marsh}, the **marshal** operation captures the linked environment between the environment pointers of its argument and the relevant mark, and the **unmarshal** operation attaches the captured environment to the current environment. We have sketched an abstract machine semantics for the above, but leave an actual implementation for future work.

3.6 Adding Distributed Communication

We now extend λ_{marsh} just enough to show examples of the rebinding scenarios from §1, sketching a $\lambda^{\text{io}}_{\text{marsh}}$ calculus. For lack of space almost all details are omitted; we show just some examples in Fig. 5, and touch on the main points.

Two extensions are required: semantics for open terms, to admit programs that use external library calls such as *print*; and communication, to support code movement. There are many design choices in combining functional and concurrent computation. Here we adopt a simple language, just to illustrate the application of λ_{marsh} and demonstrate what is required – the exact choice of primitives is therefore rather arbitrary.

We consider parallel compositions of expressions e, each with a thread ID t. One should think of threads as partitioned among a set of machines, although that structure has been omitted from the formalisation. We suppose for simplicity that all machines provide the same external library calls, with types given by a Γ_{lib}, and that there are global channels c for communication between threads, with types given by a Δ.

The semantics defines a transition relation $P \xrightarrow{l} P'$ over configurations where the labels l are either empty, $t{:}f\ u$ for an invocation by thread t of library call $f{:}T \to T'$ from Γ_{lib}, with argument u, or $t{:}u$ for a return of value u from the OS to such an invocation. The (marshal) and (unmarshal) rules must be modified slightly to deal with external identifiers.

Communication between threads is by asynchronous message passing on typed channels c, with output and input forms $e!e'$ and $e?e'$. Only marshalled values should be communicated, so communications are typed as below.

$$\frac{\Delta,\Gamma \vdash e{:}\mathsf{Chan}\ T \qquad \Delta,\Gamma \vdash e'{:}\mathsf{Marsh}\ T}{\Delta,\Gamma \vdash e!e'{:}\mathsf{unit}} \qquad \frac{\Delta,\Gamma \vdash e{:}\mathsf{Chan}\ T \qquad \Delta,\Gamma \vdash e'{:}(\mathsf{Marsh}\ T) \to T'}{\Delta,\Gamma \vdash e?e'{:}T'}$$

Example P in Fig. 5 shows rebinding to an external *print* and an internal (application library) *here*, together delimited by *AppLib*,

Simple Update Calculus: Syntax

Integers	n		Identifiers	x, y, z	Tags	i, j, k
Types	T	$::=$	$\mathsf{int} \mid \mathsf{unit} \mid T * T' \mid T \to T'$			
Expressions	e	$::=$	$x_i \mid n \mid () \mid (e, e') \mid \pi_r e \mid \lambda x_i{:}T.e \mid e e' \mid \textbf{let } z_k{:}T = e \textbf{ in } e' \mid \textbf{letrec } z_k{:}T = \lambda x_i{:}T.e \textbf{ in } e \mid \textbf{update}$			

Simple Update Calculus: Semantics

$$\text{(upd-replace-ok)}\ \frac{S = \text{rebind}(\text{fv}(e), \text{hb}(E_3)) \text{ is defined} \qquad \text{env}(E_3) \vdash S(e){:}T \qquad \forall j.x_j \notin \text{hb}(E'_3)}{E_3.\textbf{let } x_i{:}T = u \textbf{ in } E'_3.\textbf{update} \xrightarrow{\{x \Leftarrow e\}} E_3.\textbf{let } x_i{:}T = S(e) \textbf{ in } E'_3.()}$$

Figure 6: Simple Update Calculus: λ_{update}

on a communication from the left thread to the right. It has a transition sequence with labels

$$t_1{:}\textit{print}\text{“site 1”}, \quad t_1{:}(), \quad t_2{:}\textit{print}\text{“site 2”}, \quad t_2{:}()$$

for the invocations and returns of the two external $\textit{print}$ calls.

Our rebinding calculus is powerful enough to perform customized linking, useful for implementing secure encapsulation. Example Q is similar to P but the receiver defines two marks to be linked against, $\textit{TrustedAppLib}$ and $\textit{UntrustedAppLib}$. The former is for trusted programs, whereas the latter is an ‘encapsulated context,’ which reimplements both $\textit{print}$ and $\textit{here}$ with ‘safe’ versions. The safe $\textit{print}$ prints the warning string “sandboxed: ” before any output; the safe $\textit{here}$ provides the fake “site 33” to the encapsulated code, which has no way to access the true $\textit{here}_0 =$ “site 2” binding. Which context to use is determined by the hypothetical function $\textit{trusted}$, which would take into account some security criteria, such as the origin of the message. Assuming that $\textit{trusted}()$ returns $\textit{false}$, Q has a transition sequence with labels

$$\begin{array}{l} t_1{:}\textit{print}\text{“site 1”}, \quad t_1{:}(), \\ t_2{:}\textit{print}\text{“sandboxed: ”}, \quad t_2{:}(), \quad t_2{:}\textit{print}\text{“site 33”}, \quad t_2{:}() \end{array}$$

It is worth emphasising that without delayed instantiation, rebinding in these examples would not be possible. In particular, in both cases the construct-time (let) rule would substitute out $\textit{here}_0$ in t_1 before sending the lambda-term, thus preventing a rebinding of $\textit{here}$ at the remote site.

3.7 Discussion

In this subsection we review some of the design choices embodied in λ_{marsh} and their advantages and disadvantages.

A simple alternative is to allow marshalling only of values that are in some sense closed (with a marshal-time check that they do not refer to, *e.g.*, $\textit{print}$). This would require the programmer to explicitly abstract on all the identifiers that are to be treated dynamically when constructing a value to be marshalled, and to explicitly apply to the local definitions on unmarshalling. For rebinding to a single standard library this might be acceptable, though even there notationally heavy, but for the richer usages we describe above it would be prohibitively complex. One therefore needs some form of dynamic rebinding.

To keep the semantics of local computation simple, with the normal static scoping, we choose to permit rebinding only when unmarshalling values. The most interesting question is then which variables in a value should be rebound after marshalling and unmarshalling.

The main choice is between having two classes of variable (one treated statically and one dynamically), or one class of variable, with some other way of specifying which are rebound in any particular marshal/unmarshal instance.

Two classes were used in some related systems, though not motivated by marshalling [LLMS00, LF93, Dam98, Jag94] (discussed further in §5). The disadvantages of the two-class choice are: (a) it is less flexible than our use of marks, in which different marshals and unmarshals can refer to different marks, *e.g.* in the §3.6 examples; and (b) if the types or usage-forms of the two classes differ, then changing the class of a variable would require widespread code change (if the two classes are distinguished only by their declaration-forms, this is not such a problem). Code would thus be hard to maintain.

In contrast, adding marks or changing their position is syntactically lightweight; it does not require any change to code except at marshal/unmarshal points. Moreover, it will usually be straightforward to change the let-bindings in programs that contain marks: changing let-bindings inside marks is as usual; changing them outside a mark may require corresponding changes outside other marks but no change to any **marshal** and **unmarshal** expressions. Taking one class has the disadvantage that it is not obvious from a code fragment which variables might have been rebound, but in typical cases one can simply look for enclosing marks and **marshal**s.

A further disadvantage of λ_{marsh} is that programs with many nested marks, and with marks under lambdas, can become confusing. Whether this is a problem in practice remains to be seen.

With one class one could specify the variables to be rebound either with marks or by explicitly annotating **marshal** with the set of rebindable identifiers. We believe the latter would be cumbersome in practice (with large sets of standard library identifiers). It would also be conceptually complex and difficult to implement efficiently – for example, consider a sequence of bindings, each depending on the one before, around a **marshal** that specifies that alternate bindings should be treated dynamically.

4. Simple Update Calculus: λ_d + update

We now turn from dynamic rebinding of marshalled values to the rebinding involved in dynamic update. Dynamic updating is required for long-running systems that must provide uninterrupted service – the canonical example is the telephone switch, with a complex internal state, many overlapping interactions with its environment, and a requirement for high availability. Applying updates, however, can quickly lead to confusion – particularly if they are in the form of binary patches. To ameliorate this, we would like *high-level* update primitives: with semantics expressed in terms of the source programming language rather than some abstract machine or particular compilation strategy. We show this can be done for typed CBV functional programs. Delayed instantiation is again required, now so that running code picks up any updated definitions as it executes, and applying an update involves some explicit rebinding. We design a λ_{update} calculus accordingly, again based on our λ_d semantics and with tagged identifiers. It is intended as a proof-of-concept, to demonstrate that a clean high-level semantics can be based on λ_d, rather than a complete treatment of updating, so we include only a simple update primitive. Nonetheless, the calculus is still quite expressive, and unlike other work in this area is not tied to a particular abstract machine, or to a first-order setting.

The λ_{update}-calculus is given in Fig. 6 (the λ_d rules and error rules are elided). As in §3 it is convenient to use tagged identifiers and explicitly-typed **let**s, but the types are omitted in examples. We allow the programmer to place an expression **update** at points in the code where an update could occur; defining such updating 'safe points' is useful for ensuring programs behave properly [Hic01]. The intended semantics is that this expression will block, waiting for an update (possibly null) to be fed in. An update can modify any identifier that is within its scope (at update-time), for example in

let $x_1 =$ (**let** $w_1 = 4$ **in** w_1) **in**
let $y_1 =$ **update in**
let $z_1 = 2$ **in**
(x_1, z_1)

x_1 may be modified by the update, but w_1, y_1 and z_1 may not. For simplicity we only allow a single identifier to be rebound to an expression of the same type, and we do not allow the introduction of new identifiers.

We define the semantics of the update primitive using a labelled transition system, where the label is the updating expression. For example, supplying the label $\{x \Leftarrow \pi_1(3,4)\}$ means that the nearest enclosing binding of x is replaced with a binding to $\pi_1(3,4)$. Note that updates can be expressions, not just values – after an update the new expression, if not a value, will be in redex position. Further, they can be open, with free variables that become bound by the context of the **update**.

The static typing rule for **update** is trivial, as it is simply an expression of type unit. Naturally we have to perform some type checking at run-time; this is the second condition in the transition rule in Fig. 6. Notice however, that we do not have to type-check the whole program; it suffices to check that the expression to be bound to the given identifier has the required type in the context that it will evaluate in. The other conditions of the transition rule are similarly straightforward. The first ensures that a rebinding substitution is defined, *i.e.* that the context E_3 has hole binders that are alpha-equivalent to the free variables of e. Here rebind(V, L), for a set V and list L of variables, is defined if for all $x_i \in V$ there is some j with $x_j \in L$, in which case it is the the substitution taking each such x_i to the rightmost such x_j. The third condition ensures that the binding being updated, x_i, is the closest such binding occurrence for x (notice that an equivalence class x is specified for the update, but that the closest enclosing member, x_i, of this class is chosen as the updated binding). These conditions are sufficient to ensure that the following theorem holds.

THEOREM 8 (TYPE PRESERVATION FOR UPDATES).
If $\vdash e{:}T$ *and* $e \overset{\{x \Leftarrow e'\}}{\longrightarrow} e''$ *then* $\vdash e''{:}T$

We have safety and unique decomposition results that follow the form of Theorems 1 and 3.

Our use of delayed instantiation cleanly supports updating higher-order functions. Consider the following program:

let $f_1 = \lambda y_1.(\pi_2 y_1, \pi_1 y_1)$ **in**
let $w_1 = \lambda g_1.$**let** $\_ =$ **update in** $g_1(5,6)$ **in**
let $y_1 = f_1(3,4)$ **in**
let $z_1 = w_1 f_1$ **in**
(y_1, z_1)

which contains an occurrence of **update** in the body of w_1. If, when w_1 is evaluated, we update the function f:

$$e \longrightarrow^* \xrightarrow{\{f \Leftarrow \lambda p_1.p_1\}} \longrightarrow^* u$$

we have $[\![\, u \,]\!] = ((4,3),(5,6))$. Delayed instantiation plays a key role here: with the λ_c semantics, the result would be $[\![\, u \,]\!] = ((4,3),(6,5))$; *i.e.* the update would not take effect because the g_1 in the body of w_1 would be substituted away by the (app) rule before the update occurs. Our semantics preserves both the structure of contexts and the names of variables so that updates can be expressed.

Erlang [AVWW96] has a simple update mechanism where modules can be replaced at runtime. The transition to a new module, or the continued use of the old module, is specified at each call site. A semantics for a (higher-order, typed) version of the Erlang update mechanism extended to support multiple coexisting module versions can easily be expressed using the ideas in this paper [BHSS03].

5. Related Work

5.1 Lambda Calculi

As discussed in §2.2, our approach in λ_r and λ_d of using **let**s to record the arguments of functions has some similarities to prior work on explicit substitutions [ACCL90] and on sharing in call-by-need languages [AFM$^+$95].

There are also similarities with Felleisen and Hieb's syntactic theory of state [FH92]. Their Λ_S models late (redex-time) resolution of state variables in a substitution-based system by labelling the substituted-in values with the name of the variable; assignment to a variable triggers a global replacement of all values labelled with that variable throughout the program with the new value. This is then revised to an equivalent store-based model.

5.2 Dynamic Rebinding and λ_{marsh}

Dynamic Binding Work on dynamic binding can be roughly classified along three dimensions. First, one can have either *dynamic scoping*, in which variable occurrences are resolved with respect to their dynamic environment, or *static scoping with explicit rebinding*, where variables are resolved with respect to their static environment, but additional primitives allow explicit modification of these environments. Second, one can work either with one class of variables or split into two: one treated statically and one dynamically. Third, for explicit rebinding the variables to be rebound can be specified either individually, per name, or as all those bound by a certain term context. We identify some points in this space below, and refer the reader to the surveys of Moreau and Vivas [Mor98, VF01] for further discussion.

Dynamic scoping first appeared in McCarthy's Lisp 1.0 as a bug, and has survived in most modern Lisp dialects in some form. It is there usually referred to as "dynamic binding." Lisp 1.0 had one class of variables. MIT Scheme's [MIT] `fluid-let` form and Perl's `local` declaration similarly perform dynamically-scoped rebinding of variables. Modern Lisp distinguishes at declaration time between dynamically and statically scoped variables, as formalised in the λ_d-calculus of Moreau [Mor98]. Lewis *et al.* propose to add syntactically-distinct, dynamically-scoped *implicit parameters* [LLMS00] to statically-scoped Haskell. While flexible, dynamic scoping can result in unpredictable behaviour, since variables can be inadvertently captured; this was referred to as the *downward funarg problem* in the Lisp community (to avoid this in a typed setting Lewis *et al.* forbid arguments of higher-order functions from using dynamically scoped variables).

Turning to static scoping with explicit rebinding, the *quasi-static scoping* Scheme extension of Lee and Friedman [LF93] and the λN-calculus of Dami [Dam98] both have two classes of variable with a rebinding primitive that specifies new bindings for individual variables. Jagannathan's *Rascal* language [Jag94] maintains both a static environment and a *public* environment, corresponding again to two variable classes. The *barrier*, *reify*, and *reflect* operations allow explicit manipulation of the variables bound by an entire term context.

Outside the above classification, MIT Scheme also permits explicit manipulation of *top-level* environments. Hashimoto and Ohori introduce a typed context calculus [HO01] for expressing first-class evaluation contexts within the lambda calculus. Context holes can be 'filled in' with terms having free variables which are captured by the surrounding context. This allows binding at context-application time, but does not support rebinding. It is developed in the *MobileML* language [HY00]. Garrigue [Gar95] presents a calculus based on streams that can be used to encode dynamic binding for particular, *scope-free* variables.

Locating our λ_{marsh} calculus in this space, it adopts static scoping with explicit rebinding, has a single class of variables, and supports rebinding with respect to named contexts (not of individual variables). Use of the destruct-time strategy delays variable resolution until the last possible moment to give the most useful semantics, *e.g.*, for repeatedly-mobile code. As argued in §3, we believe these choices will lead to code that is easier to write and maintain, particularly for large systems.

We conjecture that λ_{marsh} could be encoded in Rascal, and also that it could be given semantics either in an environment-passing style or using an abstract machine with concrete environments. We believe, however, that our reduction semantics, with small-step reductions over the source syntax, is more perspicuous.

Partial Continuations The context-marking operator **mark** is reminiscent of Felleisen and Friedman's [FF87] prompt operator $\#$, and **marshal/unmarshal** of their control operator $\mathcal{F}$. Their operators capture partial *continuations*, whereas our operators may be seen as capturing partial *environments*: whereas **mark** marks a *binding* context, $\#$ marks an *evaluation* context. In fact, λ_{marsh} filters the captured context to retain only the binding structure (E_2), whereas Felleisen *et al.*'s semantics exhibits the behaviour of our λ_c, eagerly substituting out bindings and leaving only the control structure (E_1) to be captured.

Another interesting connection is between abstract continuations [FWFD88], as used by Queinnec [Que93], and the reduction contexts E_3 used in our operational semantics. Each A_1 or A_2 corresponds to a frame of the continuation, except that the semantics of ACPS substitutes the A_2 binding frames away.

Gunter *et al.* [GRR95] have studied $\#$ and $\mathcal{F}$ in a typed setting. It is interesting to note that although they state a type safety result, this does not exclude the possibility that a well-typed program can get 'stuck' if an appropriate prompt does not exist (*c.f.* §3.4).

In the λ_{marsh} calculus, marks are named (not anonymous), are not bound, and are preserved by marshal/unmarshal operations. Some other choices have been investigated in the context of partial continuations by Moreau and Queinnec [MQ94, Que93].

Dynamic Linking Dynamic linking is a ubiquitous simple form of dynamic binding, allowing program bindings to be resolved either at load-time or run-time, rather than statically. However, once dynamically bound, a variable's definition is fixed, precluding rebinding for marshalling or update.

Rebinding in Distributed Calculi A number of distributed process calculi provide implicit rebinding of names, adopting interaction primitives with meanings that depend on where they are used in a location structure [CG98, SV00, RH99, Sch02, SWP99, CS00]. This allows a form of rebinding to application libraries, but these works do not address the problem of integrating this rebinding with local functional computation.

The JoCaml and Nomadic Pict languages for mobile computation [FGL$^+$96, SWP99] provide rebinding to external functions, but the details are matters of implementation, not semantically specified – though a more principled proposal for JoCaml has been made by Schmitt in a Join-calculus setting [Sch02].

5.3 Dynamic Update

There are a number of implemented systems for dynamic updating surveyed in [Hic01], notably including Erlang [AVWW96]. There is very little rigorous semantics, however. Duggan [Dug01] has a formal framework for updating types, but updating code is considered only informally, based on arguments around reference types. Gilmore *et al.* [GKW97, Wal01] have a formal description of updating, but it is centred on abstract types, and is tied to their particular abstract machine. Neither of these systems properly handles updating first-class functions. Gilmore *et al.* require that a function not be *active* when it is updated; closures in activation records are active, and cannot thus be updated. Reference-based indirections require that the types of function arguments change in a way that interacts poorly with polymorphism [Hic01].

6. Conclusions and Future Work

We have established a clean semantic foundation for dynamic rebinding and update. In particular, we

- reconciled the dynamic-rebinding need for delayed instantiation with standard CBV semantics via novel redex-time and destruct-time reduction strategies;
- introduced the λ_{marsh} calculus, providing core mechanisms for dynamic rebinding of marshalled values, with a clean destruct-time operational semantics, and argued that our design choices are appropriate for a distributed programming language;
- showed how to extend λ_{marsh} with communication and external functions, to express dynamic rebinding and secure encapsulation of transmitted code; and
- demonstrated that dynamic update of programs with higher-order functions can be expressed using similar mechanisms, by introducing the λ_{update} calculus – again with a simple destruct-time semantics.

There are several directions that are worth pursuing. Part of the motivation for this work is to cope with marshalling of values in distributed functional languages, but this paper does not deal with issues of type coherence between separately-compiled run-times. One might combine $\lambda^{\text{io}}_{\text{marsh}}$ with the hash types of [LPSW03].

The $\lambda^{\text{io}}_{\text{marsh}}$ calculus has communication on channels but not π-calculus-style new channel generation. Adding these is an interesting problem, as the usual π semantics allows scope extrusion of new-binders but for **marshal/unmarshal** we require a semantics that preserves the shape of the binding environment outside marks.

This paper has focussed on semantics for small calculi, but ultimately dynamic rebinding mechanisms should be integrated with full-scale programming languages. For ML-like languages with second-class module systems it may be natural to have **mark** only at the module level (loosely analogous to the allowing marks only between top-level λ_{marsh} **let**s). Generalising, one might wish to **marshal/unmarshal** with respect to a set of structures rather than a single mark. Libraries may need careful design to work well with mobile code, to delimit any hard-to-move OS or library state. There are obvious problems with optimised implementation of calculi with redex- or destruct-time semantics, as dynamic rebinding or update primitives invalidate general use of standard optimisations, *e.g.*, inlining, and perhaps also environment-sharing schemes. For performance it will be important to identify conditions under which such optimisations are still valid – perhaps via a characterisation of contextual equivalence for λ_{marsh}. A full implementation should obviously be carried out.

Finally, for dynamic update the λ_{update} calculus is only the beginning of a rigorous treatment. The full story must address correctness of updates with state transformation, abstract types, changing the types of variables, multi-threading, and so on.

Acknowledgments We acknowledge support from a Royal Society University Research Fellowship (Sewell), a Marconi EPSRC CASE Studentship (Stoyle), EPSRC grant GRN24872 (Wansbrough), AFRL-IFGA IAI grant AFOSR F49620-01-1-0312 (Hicks, while at Cornell University), EC FET-GC project IST-2001-33234 PEPITO, and APPSEM 2.

7. REFERENCES

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Proc. 17th POPL*, pages 31–46, 1990.

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proc. 22nd POPL*, pages 233–246, 1995.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996. 2nd ed.

[BHS+03] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. Technical Report 568, University of Cambridge Computer Lab, June 2003. `http://www.cl.cam.ac.uk/~pes20/`.

[BHSS03] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proc. 2nd International Workshop on Unanticipated Software Evolution (USE 2003)*, April 2003.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. 1st FoSSaCS, LNCS 1378*, pages 140–155, 1998.

[CS00] Tom Chothia and Ian Stark. A distributed pi-calculus with local areas of communication. In *Proc. 4th HLCL, ENTCS 41.2*, 2000.

[Dam98] Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2):201–231, 1998.

[Dug01] Dominic Duggan. Type-based hot swapping of running modules. In *Proc. 5th ICFP*, pages 62–73, 2001.

[FF87] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–219. Elsevier North-Holland, 1987.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proc. 7th CONCUR, LNCS 1119*, pages 406–421, 1996.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *ACM Conference on LISP and Functional Programming, Snowbird, Utah*, pages 52–62, July 1988.

[Gar95] Jacques Garrigue. Dynamic binding and lexical binding in a transformation calculus. In *Workshop on Functional and Logic Programming*, 1995.

[GKW97] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, The University of Edinburgh, 1997.

[GRR95] C.A. Gunter, D. Rémy, and J.G. Riecke. A generalisation of exceptions and control in ML-like languages. In *Proc. FPCA*, pages 12–23, 1995.

[Hic01] Michael Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, August 2001.

[HO01] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1-2):249–272, 2001.

[HY00] Masatomo Hashimoto and Akinori Yonezawa. MobileML: A programming language for mobile computation. In *COORDINATION, LNCS 1906*, page 198 ff., 2000.

[Jag94] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM TOPLAS*, 16(3):456–492, May 1994.

[LF93] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proc. 20th POPL*, pages 479–492, 1993.

[LLMS00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In *Proc. 27th POPL*, pages 108–118, 2000.

[LPSW03] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. ICFP 2003*, August 2003.

[MIT] MIT Scheme. `http://www.swiss.ai.mit.edu/projects/scheme/`.

[Mor98] Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, December 1998.

[MQ94] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations: A duumvirate of control operators. In *Proc. PLILP, LNCS 844*, pages 182–197, September 1994.

[Nee93] R. M. Needham. Names. In S. Mullender, editor, *Distributed Systems*, pages 315–327. Addison-Wesley, Wokingham, 2nd edition, 1993.

[Que93] Christian Queinnec. A library of high level control operators. *Lisp Pointers, ACM SIGPLAN Special Interest Publ. on Lisp*, 6(4):11–26, October 1993.

[RH99] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proc. 26th POPL*, pages 93–104, 1999.

[Sch02] Alan Schmitt. Safe dynamic binding in the join calculus. In *Proc. IFIP TCS 2002*, 2002.

[SV00] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Proc. 13th Computer Security Foundations Workshop*, pages 269–284, 2000.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, 1999.

[VF01] José Luis Vivas Frontana. *Dynamic Binding of Names in Calculi for Mobile Processes*. PhD thesis, KTH, Stockholm, March 2001.

[Wal01] Chris Walton. *Abstract Machines for Dynamic Computation*. PhD thesis, University of Edinburgh, 2001. ECS-LFCS-01-425.

Id: paper2.mng,v 1.96 2003/06/23 20:30:48 pes20 Exp
Id: common.mng,v 1.92 2003/06/23 17:22:33 pes20 Exp
Id: examples.mng,v 1.10 2003/04/05 10:11:26 pes20 Exp
Id: related.mng,v 1.49 2003/06/23 20:30:48 pes20 Exp