JFP **18** (4): 437–502, 2008.
doi:10.1017/S0956796807006600 First published online 17 October 2007 Printed in the United Kingdom

Dynamic rebinding for marshalling and update, via redex-time and destruct-time reduction

PETER SEWELL
University of Cambridge
(*e-mail:* Peter.Sewell@cl.cam.ac.uk)

GARETH STOYLE
University of Cambridge
(*e-mail:* gareth@almostlogical.org)

MICHAEL HICKS
University of Maryland, College Park
(*e-mail:* mwh@cs.umd.edu)

GAVIN BIERMAN
Microsoft Research, Cambridge
(*e-mail:* gmb@microsoft.com)

KEITH WANSBROUGH
Data Connection Ltd., Edinburgh
(*e-mail:* Keith.Wansbrough@lochan.org)

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 the 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 show how the ideas of this simple calculus extend to more real-world, module-level dynamic updating in the style of Erlang. We thereby establish primitives and a common semantic foundation for a variety of real-world dynamic rebinding requirements.

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[1]:

- 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 that 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 opens files in the `/tmp` directory only. 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. A general purpose approach to this problem is to load new code into the program and then dynamically rebind 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, and 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 and dynamic updating. We focus on 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. This paper makes three contributions:

- The central contribution is our study of delayed instantiation strategies, which admit sensible semantics when extended with rebinding. We study two such calculi, λ_r and λ_d, which embody *redex-time* and *destruct-time* instantiation

[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' (Needham 1993).

semantics, respectively, in a simply typed λ calculus. We relate them to the standard CBV operational semantics, embodied in the calculus λ_c, by proving that all three evaluation strategies are observationally equivalent.

- We illustrate mechanisms for rebinding by extending our new calculi in two ways. First, we define primitives for selectively marshalling and unmarshalling values, to be communicated between processes or saved and restored from external storage. The key problem to solve is how to handle free variables appearing in marshalled λ terms. We introduce a concept of programmer-specified *marks* to define 'mobile' and 'immobile' definitions, defining a policy of which terms should and should not be marshalled when referenced by a λ term. We sketch how marks can be used in a distributed setting (with π-calculus-like communication primitives) and can be used to implement sandboxing for untrusted code.
- Finally, we define a primitive for dynamically updating definitions in a running program. The basic approach is a rather simple extension of λ_d, with run-time system support for changing definitions according to an external specification. We also consider module-level updating in the style of the functional language Erlang (Armstrong *et al.* 1996), where we extend λ_r.

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.

The next section gives a technical overview of the main body in §3–7. The work presented here forms the foundation of subsequent research on distributed programming in Acute (Sewell *et al.* 2004, 2007), a full-scale programming language with type-safe marshalling and rebinding, and on dynamic updating in Ginseng (Stoyle *et al.* 2005; Neamtiu *et al.* 2006), an implementation for dynamically updating C programs. Relationships with this and prior work, and further discussion of the design space, are presented in §8. In §9, we conclude. Proofs of results are given in the Appendices.

This paper is a revised and extended version of the paper (Bierman *et al.* 2003a), with differences as follows: in §3 the typing and run-time error rules are included, and additional examples given; in §4, the error rules for λ_{marsh} are included and the extension with distributed communication is fleshed out with examples, typing and semantics; §7 extends the basic updating calculus to more full-featured Erlang-style (Armstrong *et al.* 1996) dynamic update (Bierman *et al.* 2003c); finally, §8 relates the work herein to our subsequent work on distributed programming and dynamic updating in full-scale programming languages.

Theorem 4 of the paper (Bierman *et al.* 2003a) asserted the observational equivalence of the three calculi λ_c, λ_r and λ_d, as a check that the latter two are essentially CBV despite their rather different evaluation strategies. After publication, we discovered a technical flaw in the original proof, and so in the technical report

(Bierman *et al.* 2003b) we stated and proved the property for a simpler language, replacing **letrec** by a non-terminating Ω (with $\Omega \longrightarrow \Omega$). A proof of the original result has now been completed, using an intricate operational correspondence argument. We summarise the main points here; the full details appear in Stoyle's PhD thesis (Stoyle 2006). The proofs of the other technical results are straightforward; we give outlines here and refer the reader to the technical report (Bierman *et al.* 2003b) for details.

2 Overview

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

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

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 the expression **let** $z = v$ **in** e transmits a function containing z to some other machine, and we have indicated 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.

Therefore, we 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 CBV nature of the calculus, however. We elaborate below on exactly what this means.

We present two reduction strategies with delayed instantiation in §3. The *redex-time* (λ_r) semantics resolves identifiers when in redex position. While this is clean and simple, it can be 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_r and 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 §4, which contains primitives for packaging a value so 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

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

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 are done by expressions

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

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** $\Gamma\ u$), and can be rebound. For example:

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

(Here the 1 on x_1 and y_1 is an α-varying *tag*, see §4.1.) 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 of a distributed programming language that supports mobile code, in §5 we sketch an extension with concurrency, communication and external library functions, giving examples showing how wrappers for encapsulating untrusted code can be expressed. We also sketch an implementation strategy. Later work (Sewell *et al.* 2004, 2007) has built on these ideas to provide a full-scale prototype distributed programming language, Acute, and the HashCaml extension of OCaml with type-safe marshalling (Billings *et al.* 2006).

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 §6. 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. Any identifier in scope at the update point can be rebound to an expression that may mention identifiers in scope at its binding point. 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. We define what it means for an update to be well typed with respect to a program; applying well-typed updates preserves typing. A benefit of λ_d is that it simply and cleanly supports updating higher-order functions, often ignored in past work. In §7, we expand λ_{update} to develop a model of updating in the style of Erlang (Armstrong *et al.* 1996) and illustrate its utility with some examples. Later work (Stoyle *et al.* 2005; Neamtiu *et al.* 2006) has built on the ideas of λ_{update} to study dynamic update of C-like languages.

3 Cell-by-value λ calculus revisited

This section reconsiders the CBV λ calculus, exploring refined operational semantics that instantiate identifiers at different times. We take a standard syntax:

Identifiers	x, y, z			
	n			
Types	T	$::=$	$\mathsf{int} \mid \mathsf{unit} \mid T * T' \mid T \to T'$	
Expressions	e	$::=$	$z \mid n \mid () \mid (e, e') \mid \pi_r\ e$	$r \in \{1, 2\}$
		$\mid$	$\lambda z{:}T.e \mid e\ e' \mid \mathbf{let}\ z = e\ \mathbf{in}\ e'$	
		$\mid$	$\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ e'$	

Expressions are taken up to the usual α-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 containing at most one such for any z. The (standard) typing rules are given in Figure 1.

3.1 Construct-time

The standard semantics, here called the *construct-time* semantics, is recalled at the top of Figure 2. We define a small-step reduction relation $e \longrightarrow e'$, using evaluation contexts E, and a run-time-error predicate $e\ \mathsf{err}$ defined in Figure 4. Context composition and application are both written with a dot, for example, $E.E'$ and $E.e$, instead of the usual heavier brackets $E[e]$. Standard capture-avoiding substitution of e for z in e' is written $\{e/z\}e'$. 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.

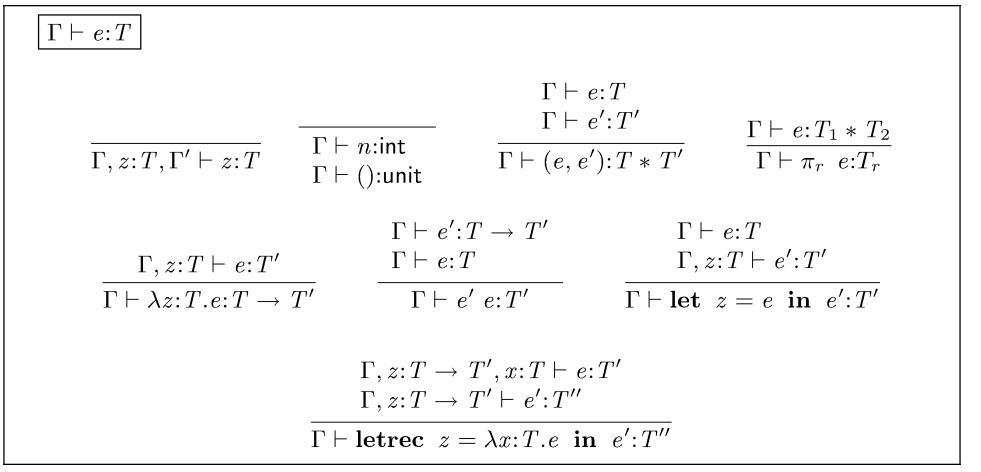

Fig. 1. Lambda calculi—typing.

3.2 Redex-time

The redex-time and destruct-time semantics are also shown in Figure 2. 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, for example,

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

This is illustrated in Example (1) in Figure 3, contrasted 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; we explain the reason for this below.

Example (2) in Figure 3 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. To allow such reduction under **let**s, we must define, in addition to the *atomic evaluation contexts* A we had above (here A_1), *binding contexts* $A_2 ::= \mathbf{let}\ z = u\ \mathbf{in}\ \_$. Reduction is closed under both, via the definition of *reduction contexts* E_3. Pure binding contexts E_2 are required to state the (inst) and (instrec) rules; pure evaluation contexts E_1 are not used, but defined here for comparison. Redex-time variable instantiation is handled with the (inst) rule, which instantiates an occurrence of the identifier z in redex position with the innermost enclosing **let** that binds that identifier. The side condition $z \notin \mathrm{hb}(E_3)$ ensures that the correct binding of z is used. Here $\mathrm{hb}(E)$ denotes the list of identifiers that bind around the hole of a context E, defined as

$$\begin{array}{lcl}
\mathrm{hb}(_) & = & [] \\
\mathrm{hb}(E.(\mathbf{let}\ z = e\ \mathbf{in}\ _)) & = & \mathrm{hb}(E), z \\
\mathrm{hb}(E.(\mathbf{letrec}\ z = \lambda x:T.e\ \mathbf{in}\ _)) & = & \mathrm{hb}(E), z \\
\mathrm{hb}(E.A) & = & \mathrm{hb}(E) \quad \text{for any other context } A
\end{array}$$

We overload $\in$ for lists. The other side condition in the (inst) rule, $\mathrm{fv}(u) \notin z, \mathrm{hb}(E_3)$, which can always be achieved by α-conversion, prevents identifier capture, making

Construct-time λ_c

Values	v	::=	$n \mid () \mid (v, v') \mid \lambda z{:}T.e$
Atomic evaluation contexts	A	::=	$(\_, e) \mid (v, \_) \mid \pi_r\ \_ \mid \_\ e \mid v\ \_ \mid \mathbf{let}\ z = \_\ \mathbf{in}\ e$
Evaluation contexts	E	::=	$\_ \mid E.A$

$$\frac{e \longrightarrow e'}{E.e \longrightarrow E.e'}$$

(proj)	$\pi_r\,(v_1, v_2)$	$\longrightarrow$	v_r
(app)	$(\lambda z{:}T.e)v$	$\longrightarrow$	$\{v/z\}e$
(let)	$\mathbf{let}\ z = v\ \mathbf{in}\ e$	$\longrightarrow$	$\{v/z\}e$
(letrec)	$\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ e'$	$\longrightarrow$	$\{\lambda x{:}T.\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ e/z\}e'$ if $z \neq x$

Redex-time λ_r

Values	u	::=	$n \mid () \mid (u, u') \mid \lambda z{:}T.e \mid \mathbf{let}\ z = u\ \mathbf{in}\ u' \mid \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ u$
Atomic evaluation contexts	A_1	::=	$(\_, e) \mid (u, \_) \mid \pi_r\ \_ \mid \_\ e \mid u\ \_ \mid \mathbf{let}\ z = \_\ \mathbf{in}\ e$
Atomic bind contexts	A_2	::=	$\mathbf{let}\ z = u\ \mathbf{in}\ \_ \mid \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ \_$
Evaluation contexts	E_1	::=	$\_ \mid E_1.A_1$
Bind contexts	E_2	::=	$\_ \mid E_2.A_2$
Reduction contexts	E_3	::=	$\_ \mid E_3.A_1 \mid E_3.A_2$

$$\frac{e \longrightarrow e'}{E_3.e \longrightarrow E_3.e'}$$

(proj)	$\pi_r\,(E_2.(u_1, u_2))$	$\longrightarrow$	$E_2.u_r$
(app)	$(E_2.(\lambda z{:}T.e))u$	$\longrightarrow$	$E_2.\mathbf{let}\ z = u\ \mathbf{in}\ e$ if $\mathrm{fv}(u) \notin \mathrm{hb}(E_2)$
(inst)	$\mathbf{let}\ z = u\ \mathbf{in}\ E_3.z$	$\longrightarrow$	$\mathbf{let}\ z = u\ \mathbf{in}\ E_3.u$ if $z \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(u) \notin z, \mathrm{hb}(E_3)$
(instrec)	$\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ E_3.z$	$\longrightarrow$	$\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ E_3.\lambda x{:}T.e$ if $z \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(\lambda x{:}T.e) \notin \mathrm{hb}(E_3)$

Destruct-time λ_d

Values	u	::=	$n \mid () \mid (u, u') \mid \lambda z{:}T.e \mid \mathbf{let}\ z = u\ \mathbf{in}\ u' \mid \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ u \mid z$
Atomic evaluation contexts	A_1	::=	$(\_, e) \mid (u, \_) \mid \pi_r\ \_ \mid \_\ e \mid u\ \_ \mid \mathbf{let}\ z = \_\ \mathbf{in}\ e$
Atomic bind contexts	A_2	::=	$\mathbf{let}\ z = u\ \mathbf{in}\ \_ \mid \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ \_$
Evaluation contexts	E_1	::=	$\_ \mid E_1.A_1$
Bind 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	::=	$\pi_r\ \_ \mid \_\ u$

$$\frac{e \longrightarrow e'}{E_3.e \longrightarrow E_3.e'}$$

(proj) $\pi_r\,(E_2.(u_1, u_2)) \longrightarrow E_2.u_r$

(app) $(E_2.(\lambda z{:}T.e))u \longrightarrow E_2.\mathbf{let}\ z = u\ \mathbf{in}\ e$ if $\mathrm{fv}(u) \notin \mathrm{hb}(E_2)$

(inst-1)

$\mathbf{let}\ z = u\ \mathbf{in}\ E_3.R.E_2.z \longrightarrow \mathbf{let}\ z = u\ \mathbf{in}\ E_3.R.E_2.u$ if $z \notin \mathrm{hb}(E_3, E_2)$ and $\mathrm{fv}(u) \notin z, \mathrm{hb}(E_3, E_2)$

(inst-2)

$R.E_2.\mathbf{let}\ z = u\ \mathbf{in}\ E_2'.z \longrightarrow R.E_2.\mathbf{let}\ z = u\ \mathbf{in}\ E_2'.u$ if $z \notin \mathrm{hb}(E_2')$ and $\mathrm{fv}(u) \notin z, \mathrm{hb}(E_2')$

(instrec-1)

$\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ E_3.R.E_2.z \longrightarrow \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ E_3.R.E_2.\lambda x{:}T.e$ if $z \notin \mathrm{hb}(E_3, E_2)$ and $\mathrm{fv}(\lambda x{:}T.e) \notin \mathrm{hb}(E_3, E_2)$

(instrec-2)

$R.E_2.\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ E_2'.z \longrightarrow R.E_2.\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ E_2'.\lambda x{:}T.e$ if $z \notin \mathrm{hb}(E_2')$ and $\mathrm{fv}(\lambda x{:}T.e) \notin \mathrm{hb}(E_2')$

Fig. 2. Three call-by-value λ calculi.

	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	$\mathbf{let}\ \ z = 8\ \ \mathbf{in}\ \ 7$	$\mathbf{let}\ \ z = 8\ \ \mathbf{in}\ \ 7$
(2)	$\mathbf{let}\ \ x = 5\ \ \mathbf{in}\ \ \pi_1\,(x, x)$	$\mathbf{let}\ \ x = 5\ \ \mathbf{in}\ \ \pi_1\,(x, x)$	$\mathbf{let}\ \ x = 5\ \ \mathbf{in}\ \ \pi_1\,(x, x)$
$\longrightarrow$	$\pi_1\,(5, 5)$	$\mathbf{let}\ \ x = 5\ \ \mathbf{in}\ \ \pi_1\,(5, x)$	$\mathbf{let}\ \ x = 5\ \ \mathbf{in}\ \ x$
$\longrightarrow$	5	$\mathbf{let}\ \ x = 5\ \ \mathbf{in}\ \ \pi_1\,(5, 5)$	
$\longrightarrow$		$\mathbf{let}\ \ x = 5\ \ \mathbf{in}\ \ 5$	
(3)	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = x\ \ \mathbf{in}\ \ \pi_1\ \ y$	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = x\ \ \mathbf{in}\ \ \pi_1\ \ y$	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = x\ \ \mathbf{in}\ \ \pi_1\ \ y$
$\longrightarrow$	$\mathbf{let}\ \ y = (5, 6)\ \mathbf{in}\ \ \pi_1\ \ y$	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = (5, 6)\ \mathbf{in}\ \ \pi_1\ \ y$	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = x\ \ \mathbf{in}\ \ \pi_1\ \ x$
$\longrightarrow$	$\pi_1\,(5, 6)$	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = (5, 6)\ \mathbf{in}\ \ \pi_1\,(5, 6)$	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = x\ \ \mathbf{in}\ \ \pi_1\,(5, 6)$
$\longrightarrow$	5	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = (5, 6)\ \mathbf{in}\ \ 5$	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \mathbf{let}\ \ y = x\ \mathbf{in}\ \ 5$
(4)	$\pi_1\,(\pi_2\,(\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ (4, x))$	$\pi_1\,(\pi_2\,(\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ (4, x))$	$\pi_1\,(\pi_2\,(\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ (4, x))$
$\longrightarrow$	$\pi_1\,(\pi_2\,(4, (5, 6)))$	$\pi_1\,(\pi_2\,(\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ (4, (5, 6)))$	$\pi_1\,(\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \ x)$
$\longrightarrow$	$\pi_1\,(5, 6)$	$\pi_1\,(\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ (5, 6))$	$\pi_1\,(\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ (5, 6))$
$\longrightarrow$	5	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \ 5$	$\mathbf{let}\ \ x = (5, 6)\ \mathbf{in}\ \ 5$

Fig. 3. Call-by-value λ calculi examples.

E_3 and **let** $z = u$ **in** _ transparent for u. Here fv(_) denotes the set of free identifiers of an expression or context.

This and the first example both illustrate a further aspect of the redex-time calculus: values u include let-bindings of the form **let** $z = u$ **in** u'. Intuitively, this is because a value should 'carry its bindings with it', preventing otherwise stuck applications, for example, $(\lambda x{:}\mathsf{int}.x)$(**let** $z = 3$ **in** 5) or, for an example where the **let** is not garbage, $(\lambda f{:}(\mathsf{int} \to \mathsf{int}).x \quad 2)$(**let** $z = 3$ **in** $\lambda x{:}\mathsf{int}.z$). Note that identifiers are not values, so z, (z, z), and **let** $z = 3$ **in** (z, z) are not values. Values may contain free identifiers under λ's, as usual, so $\lambda x{:}\mathsf{int}.z$ is an open value and **let** $z = 3$ **in** $\lambda x{:}\mathsf{int}.z$ 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 λ; terms such as π_1(**let** $z = 3$ **in** $(4, 5)$) or, more interestingly, π_1(**let** $z = 3$ **in** $(\lambda x{:}\mathsf{int}.z, 5)$) would otherwise be stuck. The side condition for (app) can always be achieved by α-conversion; it prevents capture.

Because λ_r values may involve **let**s, some clean-up is needed to extract the usual final result, for which we define

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

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

The redex-time semantics is reminiscent of an explicit substitution (Abadi *et al.* 1990), save that here the **let** will not be percolated through the term structure, and also of the λ_{let} calculus (Ariola *et al.* 1995), 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 depend on it.

3.3 Destruct-time

The redex-time strategy is appealingly simple, but it instantiates earlier than necessary. In Example (2) in Figure 3, 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, one might want instantiate as late as possible. For example, in repeatedly mobile code, we might want to instantiate each identifier only as needed to always pick up the current local definitions. Similarly, for dynamically updateable code, we may want to delay looking up a variable as long as possible, so as to acquire the most recent version. We explore these possibilities here by introducing the *destruct-time* semantics. The choice between redex-time and destruct-time semantics remains an interesting one, however. The latter may be more complex for programmers to understand and may be harder to implement efficiently.

The λ_{update} calculus of §7 uses destruct-time semantics, but for the $\lambda^{\text{mod}}_{\text{update}}$calculus of §7, and for our later Acute and HashCaml programming languages, we revert to a redex-time semantics (for module fields, as rebinding is at the module level for these languages).

To instantiate as late as possible, while remaining CBV, 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 λ structure must be made manifest. The *destruct contexts* $R ::= \pi_r \ \_|\_ \ u$ can be seen as the outer parts of the construct-time (proj) and (app) redexes. This 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. This destruct-time semantics can be viewed as a limited form of lazy evaluation in which only naked identifiers (and not arbitrary expressions) are evaluated lazily; this restriction is important for ensuring that λ_d remain 'essentially CBV' (see §3.5).

The essential change from the redex-time semantics is that now any identifier z is a value u. 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, for example:

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

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

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

Furthermore, 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 between the **let** and z. Similarly, (inst-1) must allow bindings E_2 between the R and z, and it must allow both binding and evaluation contexts E_3 between the **let** and the R. This handles the case

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

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

Example (3) in Figure 3 illustrates (inst-1) for a program with nested binders, while Example (4) illustrates (inst-2) for a program with nested destructors. It is interesting to notice how in Example (3) the chain of instantiations is handled from outside-in for λ_r and from inside-out for λ_d.

Construct-time λ_c

(proj-err)	$E.\pi_r\ v$	err	if not exists $v_1, v_2.v = (v_1, v_2)$
(app-err)	$E.v'\ v$	err	if not exists $(\lambda z{:}T.e).v' = \lambda z{:}T.e$

Redex-time λ_r

Outermost-structure-manifest values $w ::= n \mid () \mid (u, u') \mid \lambda z{:}T.e$

(proj-err)	$E_3.\pi_r\,(E_2.w)$	err	if $\neg\ \exists\ u_1, u_2.w = (u_1, u_2)$
(app-err)	$E_3.(E_2.w)u$	err	if $\neg\ \exists(\lambda z{:}T.e).w = \lambda z{:}T.e$

Destruct-time λ_d

Outermost-structure-manifest values $w ::= n \mid () \mid (u, u') \mid \lambda z{:}T.e \mid z$

(proj-err)
$E_3.\pi_r\,(E_2.w)$ err if $\neg\ \exists\ u_1, u_2.w = (u_1, u_2)$ and $\neg\ \exists\ z\ \in \mathrm{hb}(E_3, E_2).w = z$
(app-err)
$E_3.(E_2.w)u$ err if $\neg\ \exists(\lambda z{:}T.e).w = \lambda z{:}T.e$ and $\neg\ \exists\ z\ \in \mathrm{hb}(E_3, E_2).w = z$

Fig. 4. Three call-by-value λ calculi—error rules.

3.4 Soundness properties

This subsection gives basic properties of our various λ calculi: sanity checks to confirm that our definitions are coherent. The proofs for the redex-time and destruct-time calculi are slightly different to those for the usual λ_c calculus, but they are essentially routine. Detailed proofs can be found in the technical report (Bierman *et al.* 2003b).

First, we recall the important unique decomposition property of evaluation contexts for λ_c, essentially as in Felleisen and Friedman (1987), namely, that any expression is either a value, an error, or has a unique decomposition into an evaluation context and a redex. Generalising this to the more subtle evaluation contexts of λ_r and λ_d, we have the following.

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.

The destruct-time error rules defining e err, given in Figure 4, must include cases for identifiers in destruct contexts that are not bound by enclosing **let**s and so are not instantiable, giving stuck non-value expressions. Determinacy is a trivial corollary. We also have standard 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 $\Gamma \vdash e'{:}T$.

As λ_r and λ_d involve only single instantiations, not general substitution, the proofs of those results do not need the usual substitution lemma. Instead, they rely on straightforward lemmas, inverting the type judgement for terms of the form $E_2.e$ and $E_3.e$.

Theorem 3 (*Safety for λ_c, λ_r and λ_d*)
If $\vdash e:T$, then $\neg(e\ \mathsf{err})$.

The normal progress result, that if $\vdash e:T$, then either e is a value or there exists e' such that $e \longrightarrow e'$, is an immediate corollary of this and Theorem 1 above.

3.5 Contextual equivalence properties

In the previous sections, we have defined two variants of the CBV λ calculus with delayed instantiation reduction strategies. In this section, we show that whilst the reduction strategies are different, they are consistent with each other and with the CBV λ calculus (λ_c). By this we mean that the contextual equivalence relations for λ_c, λ_r and λ_d coincide. Contextual equivalence for λ_c is standard and repeated for completeness below.

Definition 1 (*Contextual equivalence for λ_c*)
Expressions e and e' are *contextually equivalent* in λ_c, written $e \overset{\text{ctx}}{=}_c e'$, if and only if for all C such that $\vdash C[e]:\mathsf{int}$ and $\vdash C[e']:\mathsf{int}$ the following hold:

i. if $C[e] \longrightarrow_c^* n$, then $C[e'] \longrightarrow_c^* n$
ii. if $C[e'] \longrightarrow_c^* n$, then $C[e] \longrightarrow_c^* n$

For the delayed instantiation calculi, we define contextual equivalence to relate terms that reduce to values that collapse to identical terms under $[\![-]\!]$ (where $[\![-]\!]$ is the value-collapsing function defined earlier). In other words, the environment is substituted away before terms are compared at the end of the computation.

Definition 2 (*Contextual equivalence for λ_r, λ_d*)
Expressions e and e' are *contextually equivalent* in $\lambda_{r/d}$, written $e \overset{\text{ctx}}{=}_{r/d} e'$, if and only if for all C such that $\vdash C[e]:\mathsf{int}$ and $\vdash C[e']:\mathsf{int}$ the following hold:

i. if $C[e] \longrightarrow_{r/d}^* v$, then $\exists\, v'.C[e'] \longrightarrow_{r/d}^* v' \wedge [\![v]\!] = [\![v']\!]$
ii. if $C[e'] \longrightarrow_{r/d}^* v$, then $\exists\, v'.C[e] \longrightarrow_{r/d}^* v' \wedge [\![v]\!] = [\![v']\!]$

To prove this coincidence of contextual equivalence relations, we first prove the following key theorem.

Theorem 4 (*Observational equivalence*)
λ_c, λ_r and λ_d are all observationally equivalent at integer type:

1. If $\vdash e:\mathsf{int}$ and $e \longrightarrow_c^* n$, then for some u we have $e \longrightarrow_{r/d}^* u$ and $[\![u]\!] = n$
2. If $\vdash e:\mathsf{int}$ and $e \longrightarrow_{r/d}^* u$, then for some n we have $e \longrightarrow_c^* n$ and $[\![u]\!] = n$

Given this result, we can show the coincidence of contextual equivalence as follows.

Theorem 5 (*Coincidence of contextual equivalence*)
$\overset{\text{ctx}}{=}_c$, $\overset{\text{ctx}}{=}_r$ and $\overset{\text{ctx}}{=}_d$ are equivalent relations.

Proof
It is sufficient to show for all e and e' that $e \overset{\text{ctx}}{=}_c e' \iff e \overset{\text{ctx}}{=}_r e'$ and $e \overset{\text{ctx}}{=}_c e' \iff e \overset{\text{ctx}}{=}_d e'$. We show just the former as the latter is similar.

case $\Longrightarrow$ **:**

First prove point (i) in the definition of $\overset{\text{ctx}}{=}_r$. Suppose

$$e \overset{\text{ctx}}{=}_c e' \tag{1}$$

$$\vdash C[e]:\mathsf{int} \tag{2}$$

$$\vdash C[e']:\mathsf{int} \tag{3}$$

$$C[e] \longrightarrow_r^* v \tag{4}$$

We prove $\exists\, v'.C[e'] \longrightarrow_r^* v' \wedge [\![v]\!] = [\![v']\!]$. By 2, 4 and observational equivalence (Theorem 4), we have $\exists\, n.C[e] \longrightarrow_c^* n \wedge n = [\![v]\!]$. By 1 and the previous fact, $C[e'] \longrightarrow_c^* n$. By 3, the previous fact and observational equivalence (Theorem 4), we have $\exists\, v''.C[e'] \longrightarrow_c^* v'' \wedge n = [\![v'']\!]$. It is immediate that $[\![v]\!] = [\![v'']\!]$, which together with the last fact proves the result.

Case (ii) is shown similarly.

case $\Longleftarrow$ **:**

Identical reasoning to the previous case.

□

The proof of both parts of Theorem 4 use the same technique: we generalise to arbitrary type and proceed to construct a bisimulation 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 bisimulation proofs is that of *instantiation normal form*. Essentially, a term is in instantiation normal form if it cannot 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 auxiliary lemmas to prove the generalised claim. The details of these proofs are very involved and given in full in Chapter 3 of Stoyle's thesis (Stoyle 2006). We give an outline and some of the details in the Appendix.

4 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. standard library calls, for example, *print*, which should be rebound at the destination;

2. application-specific location-dependent library calls, for example, routing functions, which should be rebound at the destination;
3. application code that 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 §3.3 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 occur only 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*

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

which name (the declaration parts of) contexts. Marshal and unmarshal operations

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

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 are defined outside 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 in e; 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. 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 (see, for example, Figure 12).

For simplicity, λ_{marsh} simulates communication using β-reduction (in fact, λ_d (inst) reduction), and omits treatment of standard library calls, focusing on the more interesting cases of rebinding application-specific libraries. In the next section, we sketch $\lambda^{\text{io}}_{\text{marsh}}$, which straightforwardly extends λ_{marsh} with communication and external identifiers, and discuss alternative design choices.

4.1 Syntax

The λ_{marsh} syntax and an example, discussed below, are given in Figure 5; the new semantic rules are given in Figures 6–8. The calculus requires a more elaborate treatment of α-equivalence than λ_d. There, as usual for λ calculi, we had to use α-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

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

(marshal) $\longrightarrow$

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

(inst-1) $\longrightarrow$

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

(unmarshal) $\longrightarrow$

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

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

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

Fig. 5. Dynamic rebinding calculus λ_{marsh}: Syntax and example.

Values	u	::=	$n \mid () \mid (u, u') \mid \lambda x_i{:}T.e \mid \textbf{let}\ z_k{:}T = u\ \textbf{in}\ u'$
		|	$\textbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \textbf{in}\ u \mid z_i$
		|	$\textbf{mark}\ M\ \textbf{in}\ u \mid \textbf{marshalled}\ \Gamma\ u$
Atomic evaluation contexts	A_1	::=	$(\_, e) \mid (u, \_) \mid \pi_r\ \_ \mid \_\ e \mid (\lambda x_i{:}T.e)\ \_ \mid \textbf{let}\ z_k{:}T = \_\ \textbf{in}\ e$
		|	$\textbf{marshal}\ M\ \_ \mid \textbf{unmarshal}\ M\ \_$
Atomic bind/mark contexts	A_2	::=	$\textbf{let}\ z_k{:}T = u\ \textbf{in}\ \_ \mid \textbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \textbf{in}\ \_$
		|	$\textbf{mark}\ M\ \textbf{in}\ \_$
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	::=	$\pi_r\ \_ \mid \_\ u \mid \textbf{unmarshal}\ M\ \_$

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 and $\rightharpoonup$ replacing $\longrightarrow$. These reductions are closed under E_3 by the rule below, whereas the (marshal) and (unmarshal) rules below are global.

(marshal)
$E_3.\textbf{mark}\ M.E_3'.\textbf{marshal}\ M\ u \longrightarrow E_3.\textbf{mark}\ M.E_3'.\textbf{marshalled}\ (\mathrm{env}(E_3))\ (\mathrm{bindmark}(E_3').u)$
if $\mathrm{dhb}(E_3)$ and **mark** M not around $\_$ in E_3'

(unmarshal)
$E_3.\textbf{mark}\ M.E_3'.\textbf{unmarshal}\ M.E_2.\textbf{marshalled}\ \Gamma\ u \longrightarrow E_3.\textbf{mark}\ M.E_3'.S(u)$
if $\mathrm{dhb}(E_3)$, $\mathrm{dhb}(E_3', \mathrm{hb}(E_3))$, $S = \mathrm{rebind}(\Gamma, \mathrm{env}(E_3))$ is defined,
and **mark** M not around $\_$ in E_3'.

$$\frac{e \rightharpoonup e'}{E_3.e \longrightarrow E_3.e'}$$

Fig. 6. Dynamic rebinding calculus λ_{marsh}: Semantics.

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-conversion changes only the tags; tags for different identifiers lie in different namespaces, so for example,

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

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.

4.2 Example

As an example, consider the expression on the left of Figure 5. 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 *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

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

Define the list of hole-binders of E_3, written $\mathrm{hb}(E_3)$, by:

$$\begin{array}{lcl}
\mathrm{hb}(_) & = & [] \\
\mathrm{hb}(E_3.A_1) & = & \mathrm{hb}(E_3) \\
\mathrm{hb}(E_3.(\mathbf{let}\ \ z_k{:}T = u\ \ \mathbf{in}\ \ _)) & = & \mathrm{hb}(E_3), z_k \\
\mathrm{hb}(E_3.(\mathbf{letrec}\ \ z_k{:}T' = \lambda x_i{:}T.e\ \ \mathbf{in}\ \ _)) & = & \mathrm{hb}(E_3), z_k \\
\mathrm{hb}(E_3.(\mathbf{mark}\ \ M\ \ \mathbf{in}\ \ _)) & = & \mathrm{hb}(E_3)
\end{array}$$

Define the bind and mark components of a context E_3, discarding the evaluation context components:

$$\begin{array}{lcl}
\mathrm{bindmark}(_) & = & _ \\
\mathrm{bindmark}(E_3.A_1) & = & \mathrm{bindmark}(E_3) \\
\mathrm{bindmark}(E_3.A_2) & = & \mathrm{bindmark}(E_3).A_2
\end{array}$$

Define the list of typed hole-binders of E_3, written $\mathrm{env}(E_3)$, by:

$$\begin{array}{lcl}
\mathrm{env}(_) & = & [] \\
\mathrm{env}(E_3.A_1) & = & \mathrm{env}(E_3) \\
\mathrm{env}(E_3.(\mathbf{let}\ \ z_k{:}T = u\ \ \mathbf{in}\ \ _)) & = & \mathrm{env}(E_3), (z_k{:}T) \\
\mathrm{env}(E_3.(\mathbf{letrec}\ \ z_k{:}T' = \lambda x_i{:}T.e\ \ \mathbf{in}\ \ _)) & = & \mathrm{env}(E_3), (z_k{:}T') \\
\mathrm{env}(E_3.(\mathbf{mark}\ \ M\ \ \mathbf{in}\ \ _)) & = & \mathrm{env}(E_3)
\end{array}$$

Say $\mathrm{dhb}(E_3)$ iff the list $\mathrm{hb}(E_3)$ contains no two equal elements. For such E_3 write $\mathrm{env}(E_3)$ for the obvious type environment.

Define a generalisation of the $\mathrm{dhb}(\_)$ predicate as follows. For a set X of (identifier,tag) pairs take $\mathrm{dhb}(E_2, X)$ to be the least such that

- $\mathrm{dhb}(\_, X)$
- $\mathrm{dhb}(E_2, X) \wedge z_k \notin \mathrm{hb}(E_2) \cup X \implies \mathrm{dhb}(E_2.\mathbf{let}\ \ z_k{:}T = u\ \ \mathbf{in}\ \ \_, X)$
- $\mathrm{dhb}(E_2, X) \wedge z_k \notin \mathrm{hb}(E_2) \cup X \implies \mathrm{dhb}(E_2.\mathbf{letrec}\ \ z_k{:}T' = \lambda x_i{:}T.e\ \ \mathbf{in}\ \ \_, X)$
- $\mathrm{dhb}(E_2, X) \implies \mathrm{dhb}(E_2.\mathbf{mark}\ \ M\ \ \mathbf{in}\ \ \_, X)$

and define $\mathrm{dhb}(E_3, X)$ by similar clauses together with $\mathrm{dhb}(E_3, X) \implies \mathrm{dhb}(E_3.A_1, X)$.

Define a substitution from variables in $\mathrm{dom}(\Gamma)$ to the rightmost x_j in L if types correspond:

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

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

(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 $x_j \in x_J$).

Fig. 7. Dynamic rebinding calculus λ_{marsh}: Auxiliary functions.

Outermost-structure-manifest values:
$w ::= n \mid () \mid (u, u') \mid \lambda z{:}T.e \mid z_k \mid \textbf{marshalled}\ \Gamma\ u$

(proj-err)	$E_3.\pi_r\ (E_2.w)$	err
	if $\neg\ \exists\ u_1, u_2.w = (u_1, u_2)$ and $\neg\ \exists\ z_k\ \in \mathrm{hb}(E_2, E_3).w = z_k$	
(app-err)	$E_3.(E_2.w)u$	err
	if $\neg\ \exists(\lambda x_i{:}T.e).w = \lambda x_i{:}T.e$ and $\neg\ \exists\ z_k\ \in \mathrm{hb}(E_2, E_3).w = z_k$	
(grab-err)	$E_3.\textbf{marshal}\ M\ u$	err′
	if **mark** M not around _ in E_3	
(ungrab-err1)	$E_3.\textbf{unmarshal}\ M.E_2.w$	err
	if $\neg\ \exists\ u, \Gamma.w = \textbf{marshalled}\ \Gamma\ u$ and $\neg\ \exists\ z_k\ \in \mathrm{hb}(E_2, E_3).w = z_k$	
(ungrab-err2)	$E_3.\textbf{unmarshal}\ M.E_2.\textbf{marshalled}\ \Gamma\ u$	err′
	if **mark** M not around _ in E_3	
(ungrab-err3)	$E_3.\textbf{mark}\ M.E_3'.\textbf{unmarshal}\ M.E_2.\textbf{marshalled}\ \Gamma\ u$	err′
	if $\mathrm{rebind}(\Gamma, \mathrm{env}(E_3))$ is not defined	

Fig. 8. Dynamic rebinding calculus λ_{marsh}: Error rules.

This **marshalled** $\Gamma\ u$ 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 $\mathrm{fv}(u)$ as bound by Γ in **marshalled** $\Gamma\ u$. This is emphasised in the example by showing a y_0 α-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 **let** z_1 is redundant but in more complex cases it would not be, for example, 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 $(7, 3)$ (and 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 $(6, 3)$.

4.3 Semantics

The operational rules for λ_{marsh} are given in Figure 6. 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 function to extract the bind and mark components of a context E_3, discarding the evaluation context components. This and other auxiliary functions are collected in Figure 7. bindmark(_) records the marks as well as the let-bindings so that uses of **marshal** and **unmarshal** within u will behave properly. The predicate $\mathrm{dhb}(E_3)$ holds iff the hole-binders of E_3 are all distinct (which can always be made so by α-conversion). The auxiliary $\mathrm{env}(E_3)$ extracts the type environment of the hole-binders of E_3, so they can be recorded in the **marshalled** value. We record the full type environment $\mathrm{env}(E_3)$, not just its

restriction to $\mathrm{fv}(u)$, as, while the latter would be more liberal (more unmarshals would succeed) we believe it 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 depending on the binders above the mark used for marshalling. Depending on how marshalling and marks are used, however, it is possible that the extra liberality is essential in practice, and also that this semantics leads to unacceptable accumulation of garbage type environments—a topic for future work.

The (unmarshal) rule rebinds the $\mathrm{fv}(u)$ to the let-binders in E_3 around the nearest enclosing **mark** M, using the auxiliary function $\mathrm{rebind}(\_,\_)$ to construct the appropriate substitution. Here $\mathrm{dhb}(E_3', \mathrm{hb}(E_3))$ holds iff the hole-binders of E_3' are distinct from each other and from all the variables in $\mathrm{hb}(E_3)$ (always possible by α-conversion). We define $\mathrm{rebind}(\Gamma, L)$, for a type environment Γ and list of typed hole-binders L, as a substitution taking each x_i in $\mathrm{dom}(\Gamma)$ to the rightmost x_j in L. 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.

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 apply if the argument to **unmarshal** is of the form **marshalled** $\Gamma\ u$, so the destruct contexts must include **unmarshal** M $\_$.

4.4 Typing and run-time errors

Figure 8 partitions the possible run-time errors for λ_{marsh} into two classes, e err and e err'. The first describes the usual projection/application errors, together with unmarshalling of values not of the form **marshalled** $\Gamma\ u$. Errors e err$'$ have a more dynamic nature, describing cases such as when a **marshal** or an **unmarshal** refers to a mark that is not in scope, or when at (unmarshal)-time the environment does not have the required binders at the correct types. We can define a simple type system to exclude all of the err error cases by extending the standard simple type system (Figure 1) with a type Marsh T of marshalled type T values, and rules

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

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

Because errors err' are not excluded, a full language would most likely raise catchable exceptions, thereby allowing code to dynamically check the presence of resources.

Ideally, 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.

4.5 Soundness properties

Soundness properties for λ_{marsh} are similar to those for the λ calculi of the prior section. Proofs may be found in the technical report (Bierman *et al.* 2003b).

Theorem 6 (*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.

Proof
The proof is by induction over (possibly open) λ_{marsh} expressions e. □

Theorem 4.1 (*Type preservation for* λ_{marsh})
If $\vdash e:T$ and $e \longrightarrow e'$, then $\vdash e':T$.

Theorem 4.2 (*Partial safety for* λ_{marsh})
If $\vdash e:T$, then $\neg(e$ err$)$.

4.6 Discussion

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

A simple alternative to rebinding is to allow marshalling of only those values that are in some sense closed (with a marshal-time check that they do not refer to, for example, *print*). This would require the programmer to explicitly abstract 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 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 (Lee & Friedman 1993; Jagannathan 1994; Dami 1998; Lewis *et al.* 2000) (discussed further in §8). The disadvantages of the two-class choice are: (a) it is less flexible in comparison with our use of marks, in which different marshals and unmarshals can refer to different marks, for example, in the examples of §5; 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 by their declaration-forms only, 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 λ's, can become confusing. 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.

An alternative to marks, which implicitly define the variables to be rebound, would be to specify the variables directly as an annotation to **marshal**. 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 as in:

$$\begin{array}{l}\textbf{let}\ \ w = 1\ \ \textbf{in}\\ \textbf{let}\ \ x = (w, 2)\ \textbf{in}\\ \textbf{let}\ \ y = (x, 3)\ \textbf{in}\\ \textbf{let}\ \ z = (y, 4)\ \textbf{in}\\ \textbf{marshal}*\ [z, x]e\end{array}$$

The **marshal**$*$ specifies that any references to z and x in e should be treated dynamically—but then there is no obviously satisfactory semantics for y.

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 would 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. There are obvious problems with optimised implementation of calculi with redex- or destruct-time semantics at the expression level, as dynamic rebinding or update primitives invalidate general use of standard optimisations, for example, 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}.

In addition, we have explored three actual implementations of related languages. First, in our Acute language, with a second-class module system, marks were allowed only between module declarations, with a redex-time instantiation semantics. The implementation was an interpreter corresponding closely to the operational semantics, except that it used closures and an efficient representation of evaluation contexts. Second, Billings developed a prototype adaptation of the OCaml bytecode run-time, generating this modified bytecode from a fragment of the Acute front-end (Billings 2005). Here too marks were between second-class modules; the existing OCaml bytecode implementation already essentially uses redex-time instantiation of module field references, so a reasonably efficient implementation was straightforward. Third, in the HashCaml language, the existing OCaml bytecode compiler was modified to support type-safe marshalling, but there with rebinding only to the standard library.

5 Extending λ_{marsh} with IO: $\lambda^{\text{io}}_{\text{marsh}}$

We now extend λ_{marsh} just enough to show examples of distributed rebinding scenarios from §1, defining a $\lambda^{\text{io}}_{\text{marsh}}$ calculus.

5.1 Typing and semantics

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. We present these extensions as simply as possible to illustrate the application of λ_{marsh} and demonstrate what is required—the exact choice of primitives is rather arbitrary.

The syntax is shown in Figure 9. Distributed programs are described as configurations P, composed of the null process 0; expressions e, each with a thread ID t written $t:e$; or parallel compositions of processes $P \mid P$. One should think of

$\lambda^{\mathrm{io}}_{\mathrm{marsh}}$: Syntax

Integers	n	Identifiers	x, y, z	Tags	i, j, k	Context marks	M
Strings	s	Channels	c	Thread ids	t		

Type environments	Γ		finite partial functions from (identifier,tag) pairs to types
Channel typings	Δ		finite partial functions from channels to $\mathsf{Chan}\ T$ types
Types	T	::=	$\mathsf{int} \mid \mathsf{unit} \mid T * T' \mid T \to T' \mid \mathsf{Marsh}\ T \mid \mathsf{Chan}\ T \mid \mathsf{string}$
Expressions	e	::=	$z_i \mid n \mid () \mid (e, e') \mid \pi_r\ e \mid \lambda x_i{:}T.e \mid e\ e'$
			$\mid$ **let** $z_k{:}T = e$ **in** $e' \mid$ **letrec** $z_k{:}T' = \lambda x_i{:}T.e$ **in** e'
			$\mid$ **mark** M **in** e
			$\mid$ **marshal** $M\ e \mid$ **marshalled** $\Gamma\ u \mid$ **unmarshal** $M\ e$
			$\mid$ $\mathbf{ret}_T \mid c \mid e!e' \mid e?e' \mid s$
Configurations	P	::=	$0 \mid t{:}e \mid (P \mid P')$

Binding and alpha equivalence as in λ_{marsh}.

Fig. 9. Distributed λ_{marsh}: $\lambda^{\mathrm{io}}_{\mathrm{marsh}}$—syntax.

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 global type environment, Γ_{lib}, and that there are global channels c for communication between threads. Communication between threads is by synchronous message passing on typed channels c, with output and input forms $e!e'$ and $e?e'$. Only marshalled values can be communicated. We add strings s to the language for convenience. We explain the form $\mathbf{ret}_T$ shortly.

The semantics is given in Figure 10. We define a transition relation $P \xrightarrow{l} P'$ over configurations where the labels l have three possible forms: (1) empty; (2) $t{:}f\ u$ to denote an invocation by thread t of library call $f{:}T \to T'$ from Γ_{lib}, with argument u; or (3) $t{:}u$ for a return of value u from the OS to such an invocation. The unlabelled case is for normal reduction, while the latter two describe external library calls.

Normal reduction is as in λ_{marsh}, with differences to account for communication and parallel composition. Values now include channels c and strings s. The A_1 atomic evaluation contexts include input and output, with a left-to-right evaluation order. More interestingly, the destruct contexts must include input and output on both left and right to ensure we can reduce to an explicit channel, marshalled value and λ before (comm) fires.

The (comm) rule for synchronisation simply moves values from sender to receiver (typing, shown below, ensures that channels carry values of $\mathsf{Marsh}\ T$ types only, which must be closed). The (marshal) and (unmarshal) rules are straightforward adaptions of the corresponding λ_{marsh} rules. In (marshal), note that we record Γ_{lib} in the marshalled value, thereby ensuring the marshalled value can be typed as in λ_{marsh}. The (unmarshal) rule prepends Γ_{lib} (for which we must suppose a fixed

Values	u	::=	$n \mid () \mid (u, u') \mid \lambda x_i{:}T.e \mid \textbf{let}\ z_k{:}T = u\ \textbf{in}\ u$
		$\mid$	$\textbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \textbf{in}\ u \mid z_i$
		$\mid$	$\textbf{mark}\ M\ \textbf{in}\ u \mid \textbf{marshalled}\ \Gamma\ u$
		$\mid$	$c \mid s$
Atomic evaluation contexts	A_1	::=	$(\_, e) \mid (u, \_) \mid \pi_r\ \_ \mid \_\ e \mid u\ \_$
		$\mid$	$\textbf{let}\ z_k{:}T = \_\ \textbf{in}\ e$
		$\mid$	$\textbf{marshal}\ M\ \_ \mid \textbf{unmarshal}\ M\ \_$
		$\mid$	$\_!e \mid c!\_ \mid \_?e \mid c?\_$
Atomic bind/mark contexts	A_2	::=	$\textbf{let}\ z_k{:}T = u\ \textbf{in}\ \_$
		$\mid$	$\textbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \textbf{in}\ \_$
		$\mid$	$\textbf{mark}\ M\ \textbf{in}\ \_$
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	::=	$\pi_r\ \_ \mid \_\ u \mid \textbf{unmarshal}\ M\ \_$
		$\mid$	$\_!e \mid c!\_ \mid \_?e \mid c?\_$

Rules (proj), (app), (inst-1), (inst-2), (instrec-1), and (instrec-2) are exactly as in λ_{marsh}, defining reductions $\rightharpoonup$ that may occur within any E_3 context of a thread. Rules (marshal) and (unmarshal) are adapted from the λ_{marsh} rules to take Γ_{lib} into account:

(marshal)
$t{:}E_3.\textbf{mark}\ M.E_3'.\textbf{marshal}\ M\ u$
$\longrightarrow\ t{:}E_3.\textbf{mark}\ M.E_3'.\textbf{marshalled}\ (\Gamma_{\text{lib}}, \text{env}(E_3))\ (\text{bindmark}(E_3').u)$
if $\text{dhb}(E_3, \text{dom}(\Gamma_{\text{lib}}))$ and $\textbf{mark}\ M$ not around $\_$ in E_3'

(unmarshal)
$t{:}E_3.\textbf{mark}\ M.E_3'.\textbf{unmarshal}\ M.E_2.\textbf{marshalled}\ \Gamma\ u \longrightarrow t{:}E_3.\textbf{mark}\ M.E_3'.S(u)$
if $\text{dhb}(E_3, \text{dom}(\Gamma_{\text{lib}}))$, $\text{dhb}(E_3', (\text{dom}(\Gamma_{\text{lib}}) \cup \text{hb}(E_3)))$,
$S = \text{rebind}(\Gamma, (\Gamma_{\text{lib}}, \text{env}(E_3)))$ is defined, and $\textbf{mark}\ M$ not around $\_$ in E_3'.

The rule for communication:

(comm)
$t{:}E_3.c!\textbf{marshalled}\ \Gamma\ u \mid t'{:}E_3'.c?(\lambda x_i{:}T.e)$
$\longrightarrow\ t{:}E_3.() \mid t'{:}E_3'.(\lambda x_i{:}T.e)(\textbf{marshalled}\ \Gamma\ u)$

Rules for invocations and returns of library calls:

(lib-app) $t{:}E_3.(E_2.f_i)u \xrightarrow{t:f\ [\![\ \text{bindmark}(E_3).u\ ]\!]} t{:}E_3.\textbf{ret}_{T'}$
if $f_i{:}T \to T' \in \Gamma_{\text{lib}}$ and $f_i \notin \text{hb}(E_3, E_2)$

(lib-ret) $t{:}E_3.\textbf{ret}_{T'} \xrightarrow{t:u'} t{:}E_3.u'$
if $\Gamma_{\text{lib}} \vdash u'{:}T'$ and $\text{hb}(E_3) \cap \text{dom}(\Gamma_{\text{lib}}) = \varnothing$

Rules for congruence:

$$\frac{e \rightharpoonup e'}{t{:}E_3.e \longrightarrow t{:}E_3.e'} \qquad \frac{P \xrightarrow{l} P'}{P \mid P'' \xrightarrow{l} P' \mid P''} \qquad \frac{P \equiv P' \xrightarrow{l} P'' \equiv P'''}{P \xrightarrow{l} P'''}$$

where structural congruence $\equiv$ is the least congruence over configurations satisfying $P \mid 0 \equiv P$, $P' \mid P \equiv P \mid P'$ and $(P \mid P') \mid P'' \equiv P \mid (P' \mid P'')$.

Fig. 10. Distributed λ_{marsh}: $\lambda^{\text{io}}_{\text{marsh}}$—semantics.

ordering, regarding it as a list of type assumptions $x_i{:}T$) to env(E_3) to calculate the appropriate rebinding substitution.[2]

The external library calls in Γ_{lib}, for example, $print_0{:}\mathsf{string} \to \mathsf{unit}$, have the same syntax as applications, but are handled by two separate rules, (lib-app) and (lib-ret). The (lib-app) rule reduces external calls to a place-holder $\mathbf{ret}_T$ to record that this thread is expecting a response from the OS of type T. The (lib-ret) rule allows the OS to provide that response. Both (lib-app) and (lib-ret) introduce labels annotated with the thread id performing the action, modelling the fact that IO on different machines should usually be distinguished (in practice one should work with a somewhat weaker notion of observation than this transition system, as discussed in Sewell 1997). Invocation labels $t{:}f\ u$ are not annotated with the tag i of the call, as tags should not be visible to the programmer or observer. At an invocation of an external call, we must collapse any **let**-structure of the argument to produce a concrete value (typically, one of a type not involving any function spaces). This is done in (lib-app) by the auxiliary $[\![e]\!]$ as defined for λ_d and λ_r in §3.2 but extended to the additional syntax of $\lambda^{\text{io}}_{\text{marsh}}$:

$$\begin{array}{lcl} [\![\mathbf{mark}\ M\ \mathbf{in}\ u]\!] & = & \mathbf{mark}\ M\ \mathbf{in}\ [\![u]\!] \\ [\![\mathbf{marshalled}\ \Gamma\ u]\!] & = & \mathbf{marshalled}\ \Gamma\ u \\ [\![c]\!] & = & c \\ [\![s]\!] & = & s \end{array}$$

Finally, the value returned from an external call must be well typed. The side condition $\Gamma_{\text{lib}} \vdash u'{:}T'$ of (lib-ret) allows this value to mention global channels or other library calls, liberally, though in practice one might insist that return values are closed.

The extensions to the λ_{marsh} type system (§4.4) are given in Figure 11. We use Δ as global map from channels c to types T, which we assume all have the form $\mathsf{Chan}\ T'$ for some T'. We do not state type preservation or partial safety results for $\lambda^{\text{io}}_{\text{marsh}}$; they should be straightforward (albeit tedious) adaptations of the results for λ_{marsh}.

5.2 Examples

Some examples are given in Figures 12 and 13. Example P in Figure 12 shows rebinding to an external *print* and an internal (application library) *here*, together delimited by *AppLib*, on a communication from the left thread to the right. It has a transition sequence with labels

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

for the invocations and returns of the two external *print* calls.

[2] One could easily relax our assumption that all machines provide the same external library here, though one might then wish to alter (marshal) to record only the *used* external calls—the obvious relaxation of the rule given here would prevent unmarshalling of any value from a thread with a larger standard library than that available to the unmarshaller.

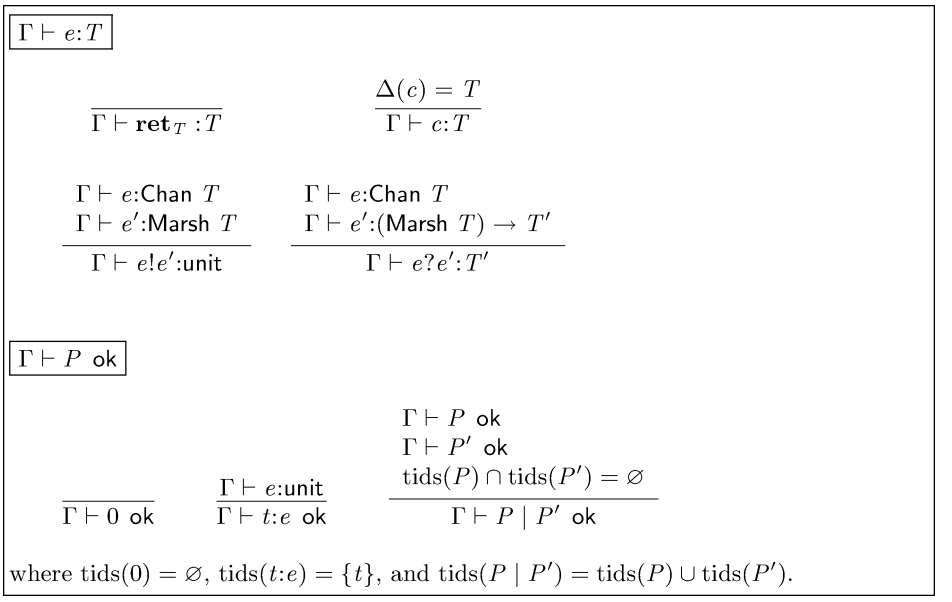

Fig. 11. Distributed λ_{marsh}: $\lambda^{\mathrm{io}}_{\mathrm{marsh}}$—typing.

Our rebinding calculus is powerful enough to perform customised linking, useful for implementing secure encapsulation. Example Q is similar to P but the receiver defines two marks to be linked against, *TrustedAppLib* and *UntrustedAppLib*. The former is for trusted programs, whereas the latter is an 'encapsulated context', which reimplements both *print* and *here* with 'safe' versions. The safe *print* prints the warning string "sandboxed: " before any output; the safe *here* provides the fake "site 33" to the encapsulated code, which has no way to access the true $here_0$ = "site 2" binding.[3] Which context to be used is determined by the hypothetical function *trusted*, which would take into account some security criteria, such as the origin of the message. Assuming that *trusted*() returns *false*, Q has a transition sequence with labels

$$t_1{:}print\text{“site 1”}, \quad t_1{:}(), \quad t_2{:}print\text{“sandboxed: ”}, \quad t_2{:}(), \quad t_2{:}print\text{“site 33”}, \quad t_2{:}()$$

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 $here_0$ in t_1 before sending the λ term, thus preventing a rebinding of *here* at the remote site.

In R, again in Figure 12, there are two communications, from t_1 to one of t_2 or t_3, and thence to the other one; rebinding of *here* and *print* occurs twice.

Example S in Figure 13 shows a use of nested marks in which marshalling copies a mark. Suppose the form of *OuterLib* (a definition of *here*) is standard on all sites,

[3] The code as given does not prevent the encapsulated code itself executing an **unmarshal** *TrustedAppLib e*. This can be protected against by redeclaring the *TrustedAppLib* mark within the conditional limit.

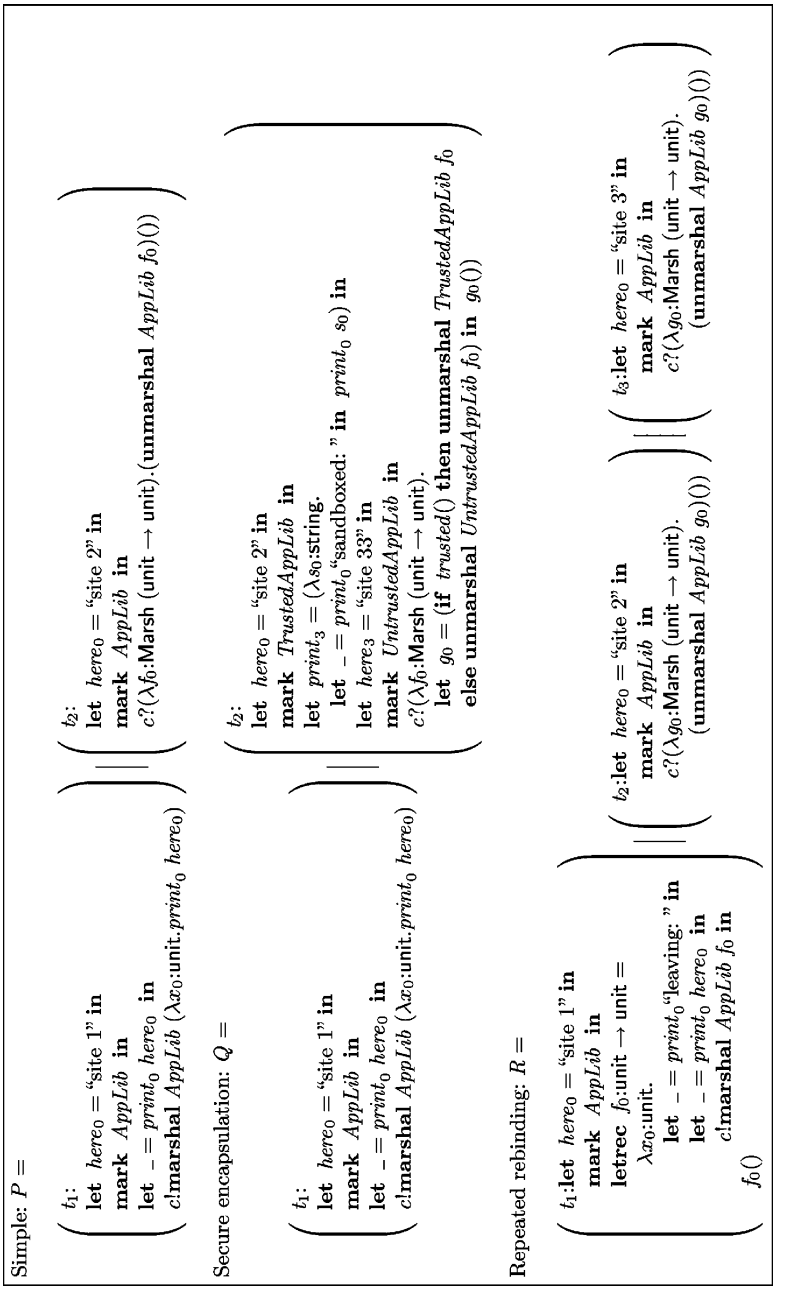

Fig. 12. Dynamic rebinding with IO and communication: $\lambda^{\mathrm{io}}_{\mathrm{marsh}}$ examples.

```
Moving Marks: S =

  t1: let here0 = "site 1 – internal" in
      mark OuterLib in
      let x0 = "internal resource (from site 1)" in
      mark InnerLib in
      let send_to_external0:(unit → unit) → unit =
        λz0:(unit → unit).c_external!marshal OuterLib z0 in
      let send_to_internal0:(unit → unit) → unit =
        λz0:(unit → unit).c_internal!marshal InnerLib z0 in
      let _ = ...use x0... in
      send_to_external0(λy0:unit.
        let _ = ...use x0... in
        let _ = send_to_internal0(λw0:unit.
          let _ = ...use x0... in
          ()))

  | t2:let here0 = "site 2 – external" in
      mark OuterLib in
      c_external?(λg0:Marsh (unit → unit).(unmarshal OuterLib g0)()

  | t3:let here0 = "site 3 – internal" in
      mark OuterLib in
      let x0 = "internal resource (from site 3)" in
      mark InnerLib in
      c_internal?(λg0:Marsh (unit → unit).(unmarshal InnerLib g0)()
```

Fig. 13. Dynamic rebinding with IO and communication: Further $\lambda^{\text{io}}_{\text{marsh}}$ examples.

whereas that of *InnerLib* (a definition of a resource x) is standard only on the sites within a particular organisation. In the example, there are two communications, from t_1 (internal) to t_2 (external) and from t_2 back to t_3 (internal). The first takes the definition of x from its departure site, but the second, returning to within the organisation, picks up the local definition of x. The three uses of x are therefore with the definitions from t_1, t_1 again and t_3.

6 A simple update calculus: λ_d + update

We now turn from dynamic rebinding of marshalled values to the rebinding involved in dynamic software updating (DSU). DSU is a technique by which a running program is patched with new code and data on-the-fly, while it runs. This is handy for systems that must provide uninterrupted service, but nonetheless require enhancements and bug fixes. One example is the telephone switch, with a complex internal state, many overlapping interactions with its environment, and a requirement for high availability. DSU is a general, software-based technique: there is no need for redundant hardware or special-purpose software architectures, and application state is naturally preserved between updated versions, so that current processing is not compromised or interrupted.

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	$::=$	$x_i \mid n \mid () \mid (e, e') \mid \pi_r\ e \mid \lambda x_i{:}T.e \mid e\ e' \mid \mathbf{let}\ z_k{:}T = e\ \mathbf{in}\ e' \mid$ $\mathbf{letrec}\ z_k{:}T = \lambda x_i{:}T.e\ \mathbf{in}\ e \mid \mathbf{update}$

Simple Update Calculus: Semantics

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

Fig. 14. Simple update calculus: λ_{update}.

Because the systems being updated are typically safety-critical, it is important to be able to reason about the meaning and possible effects of updates to ensure that the patched system operates correctly. Without care, after several updates the state of an updated system can become confusing, particularly when updates are in terms 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.

In this section and the next, we show this can be done for typed CBV functional programs. In this section, we present a simple extension to λ_d—the λ_{update} calculus—that permits let-bound variables to be rebound to new expressions. Delayed instantiation via λ_d is required so that running code picks up any rebound definitions as it executes. This calculus is simple yet powerful, and illustrates the basic ideas behind dynamic updating.

In a practical setting, updating is probably more sensible at the module level, rather than at the level of individual bindings. In the next section, we present $\lambda^{\text{mod}}_{\text{update}}$, a calculus for dynamically updating whole modules in the style of the functional language Erlang (Armstrong *et al.* 1996). For $\lambda^{\text{mod}}_{\text{update}}$, the rebinding semantics is based on λ_r, rather than λ_d. This choice makes name resolution more eager, and we discuss the trade-offs of λ_r versus λ_d in this setting. Together, the formal foundations presented here underpin our recent work on Ginseng (Stoyle *et al.* 2005; Neamtiu *et al.* 2006), a practical dynamic updating system we have built for C programs, which we discuss in §8.

6.1 The λ_{update} calculus

The λ_{update} calculus, shown in Figure 14, extends λ_d with a single primitive **update** to mark points in the code where a dynamic update could occur. As many past researchers have observed, the timing of an update is critical to ensuring its validity (Lee 1983; Frieder & Segal 1991; Gupta 1994; Hicks 2001). The *synchronous* **update** primitive, by dictating when an update can occur, makes it easier to understand the state(s) of the program which an update is applied to than the alternative

asynchronous approach, in which an update could occur at any time. Our experience is that synchronous updating makes it easier to write correct updates (Hicks 2001; Neamtiu *et al.* 2006).

Apart from this extension, the semantics of the language is exactly that of λ_d (whose operational rules are given in Figure 2 and error rules are in Figure 4). As in §4, it is convenient to use tagged identifiers and explicitly typed **let**s, but the types are omitted in examples.

The intended semantics of **update** is that evaluation will block until a dynamic update (possibly null) is available. An update can modify any identifier that is within its scope (at update-time). For example, in

$$\begin{array}{l}\textbf{let}\ \ x_1 = (\textbf{let}\ \ w_1 = 4\ \ \textbf{in}\ \ w_1)\ \textbf{in}\\ \textbf{let}\ \ y_1 = \textbf{update}\ \ \textbf{in}\\ \textbf{let}\ \ z_1 = 2\ \ \textbf{in}\\ (x_1, z_1)\end{array}$$

x_1 may be modified by the update, but w_1, y_1 and z_1 may not. For simplicity, we allow only a single identifier to be rebound to an expression of the same type, and we do not allow the introduction of new identifiers (we relax this restriction in $\lambda^{\text{mod}}_{\text{update}}$).

As shown in Figure 14, 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. Furthermore, they can be open, with free variables that become bound by the context of the **update**. This allows, among other things, computing an updated identifier's value on the basis of values in the current program (Hicks 2001).

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 Figure 14. 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, that is, that the context E_3 surrounding the to-be-updated identifier definition has hole-binders that are α-equivalent to the free variables of e. (Here, rebind(V, L) is just as in λ_{marsh}, defined in Figure 7.) The binders must be chosen from E_3 since the updating expression e is installed into that context (as opposed to the location of the **update**). In contrast to λ_{marsh}, an updateable program need not distinguish 'location-dependent' versus 'location-independent' identifiers, and thus has no need of marks to distinguish them when linking the free variables of e. The third condition ensures that the binding being updated, x_i, is the closest such binding occurrence for x with respect to the position of the update. (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.)

6.2 The role of delayed instantiation

The use of delayed instantiation via λ_d cleanly defines a model in which function applications always use the most recent function definition, whether the call is direct or indirect. Consider the following program (call it e):

$$\begin{array}{l}\textbf{let } f_1 = \lambda y_1.(\pi_2\ y_1, \pi_1\ y_1) \textbf{ in}\\ \textbf{let } w_1 = \lambda g_1.\textbf{let } \_ = \textbf{update } \textbf{ in } g_1(5,6) \textbf{ in}\\ \textbf{let } y_1 = f_1(3,4) \textbf{ in}\\ \textbf{let } z_1 = w_1\, f_1 \textbf{ in}\\ (y_1, z_1)\end{array}$$

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 via λ_d plays a key role here: with the λ_c semantics, the result would be $[\![u]\!] = ((4,3),(6,5))$; that is, 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.

With λ_r semantics, the structure of contexts and names of variables would be preserved, but instantiation would be more eager: the identifier f_1 would be instantiated prior to the call to w_1, when f_1 appears in redex position, and therefore the call to g_1 would use the original definition of f_1, not the updated one. This semantics can be more intuitive in higher-order programming in some cases. For example, it may be preferable to preserve the meaning of f when evaluating *fold f l* rather than allowing f to change in the middle. In first-order programming, the distinction between λ_r and λ_d is less apparent; the example from the prior subsection evaluates to the same answer in both semantics when x is updated.

6.3 Formal properties

The λ_{update} calculus enjoys the following formal properties. Their proofs are straightforward.

Theorem 9 (*Unique decomposition for* λ_{update})
Let e be a closed λ_{update} expression. Then, exactly one of the following holds: (1) e is a value; (2) e err; or (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.

Theorem 10 (*Type preservation for updates*)
If $\vdash e : T$ and $e \xrightarrow{\{x \Leftarrow e'\}} e''$, then $\vdash e'' : T$

Theorem 11 (*Safety for updates*)
If $\vdash e : T$, then $\neg(e\ \text{err})$.

Natural numbers	n		
Identifiers	x, y		
Module names	M		
Module component identifiers	z, f		
Versioned module names	M^n		
Types	T	::=	$\mathsf{int} \mid \mathsf{unit} \mid T * T \mid T \to T$
Module interfaces	σ	::=	$\{z_1{:}T_n, ..., z_n{:}T_n\}$
Expressions	e	::=	$n \mid () \mid (e, e') \mid \pi_r\ e \quad (r = 1, 2) \mid \lambda x{:}T.e \mid e\ e$
		|	$\mathbf{let}\ x{:}T = e\ \mathbf{in}\ e' \mid x \mid M.z \mid M^n.z \mid \mathbf{update}$
Values	v	::=	$n \mid () \mid (v, v') \mid \lambda x{:}T.e$
Module bodies	m	::=	$\{z_1{:}T_1 = v_1\ ...\ z_n{:}T_n = v_n\}$
Module sets	ms	::=	$\{\mathbf{module}\ M_1^{n_1} = m_1, . , \mathbf{module}\ M_k^{n_k} = m_k\}$
Programs	P	::=	$\mathbf{modules}\ ms\ \mathbf{in}\ e$
Atomic expr. contexts	A_1	::=	$(\_, e) \mid (v, \_) \mid \pi_r\ \_ \mid \_\ e \mid (\lambda x{:}T.e)\_$
		|	$\mathbf{let}\ x{:}T = \_\ \mathbf{in}\ e$
Expression contexts	E_1	::=	$\_ \mid A_1.E_1$
Module contexts	E_2	::=	$\mathbf{modules}\ ms\ \mathbf{in}\ \_$

We work up to α-conversion of expressions throughout, with x binding in e in an expression $\lambda x{:}T.e$ and y binding in e' in an expression $\mathbf{let}\ y{:}T = e\ \mathbf{in}\ e'$. The M_k^n of a module set and the z_i of a module body do not bind, and so are not subject to α-conversion.

Fig. 15. The $\lambda^{\text{mod}}_{\text{update}}$ calculus syntax.

7 Module-level updating: $\lambda^{\text{mod}}_{\text{update}}$

This section presents $\lambda^{\text{mod}}_{\text{update}}$, a calculus in which updates occur at the level of modules. The $\lambda^{\text{mod}}_{\text{update}}$ calculus is based roughly on the mechanisms in Erlang (Armstrong *et al.* 1996), in which the transition to a new module, or the continued use of the old module, is specified at each call site. This section is based on an earlier formalism presented at a workshop (Bierman *et al.* 2003c). We present syntax and semantics first, and then some detailed examples.

7.1 Syntax

Figure 15 shows the syntax of the language, which is basically a simply typed, CBV λ calculus with two extensions: (1) a simple module system with novel variable lookup rules, and (2) an **update** primitive that allows loading a new version of a module during program execution.

A *program* P consists of a mutually recursive set ms of module declarations and an expression e to evaluate. Module declarations are of the form $\mathbf{module}\ M^n = m$, where M is a module name, n is a *version number* and m is a module body. (Note that the version superscript n is part of the abstract syntax of programs, while a subscript k on a module name—or a variable or expression for that matter—as

in $M_k^{n_k}$, is used only to notate enumerations.) Many different versions of the same module can coexist in a program, but each pair of a module name and a version number is unique. In turn, a module body m is a collection of bindings of values for module component identifiers, written $z : T = v$.

Expressions e are as in λ_{update}, including **update** and two new forms for accessing module members. Here, the **update** primitive is used to update a module with a new version, or insert a new module. To allow staged transitions from old to new code, we allow flexible access to module components: to access the z component of a module named M, one can write either $M.z$, which will use the newest version of the module M, or $M^n.z$ (for some n), which uses version n of the code. This semantics is analogous to the semantics of Erlang, but is slightly more general. In particular, Erlang requires *all* references to an external module to invoke the newest version of the code, while internal references can be either to the 'current' version (i.e. the same version of the module as the code making the call) or to the newest version.

7.2 Semantics and typing

Figure 16 presents the dynamics of the calculus. We define a small-step reduction relation $P \longrightarrow P'$, using evaluation contexts E_1 for expressions and E_2 for programs. The rules for (let), (app) and (proj) are standard, while the remaining three rules describe accessing module bindings and updating module definitions.

Module component identifiers are resolved in the style of λ_r. In particular, the (ver) rule will resolve the component identifier z from version n of module M when the expression $M^n.z$ appears in redex position. Similarly, the (unver) rule handles the $M.z$ case, with the difference being that the most recent version of module M is used. Both rules are similar to the λ_r (inst) rule in Figure 2.[4] The semantics of $\lambda_{\text{update}}^{\text{mod}}$ is simplified from that of λ_r as we are concerned only with delayed instantiation of module names, not arbitrary identifiers. This allows us to use the standard (app) rule, for example, rather than λ_r's (app) rule, which introduces a let-binding for possible rebinding later.

The (update) rule uses labelled transitions $P \xrightarrow{M^n=m} P'$, which loads version n of module M (having body m) into the program, assuming that type safety is not compromised and as long as n is greater than any existing version of M. Any unversioned existing references to M in the code will now refer to the newly loaded module.

We can now look at an example update. In the following, take

$$ms \equiv \{\textbf{module } M^0 = \{f = \lambda x : \mathsf{unit}.\textbf{let } y : \mathsf{unit} = \textbf{update } \textbf{in } M.z \\ z = 3\}\}$$

[4] We could use the more standard substitution semantics for $M^n.z$, but this is unnecessary as the semantics of **update** prevents replacing an extant module version. We therefore use the delayed lookup semantics for symmetry in both cases.

$\boxed{P \xrightarrow{M^n=m} P'}$ (applying a module update to a program)

$$\text{(update)}\quad \frac{\varnothing \vdash \textbf{modules}(ms \cup \{\textbf{module}\ M^n = m\})\ \textbf{in}\ E_1.() : T \qquad n > \text{maxversion}(M, ms)}{\textbf{modules}\ ms\ \textbf{in}\ E_1.\textbf{update} \xrightarrow{M^n=m} \textbf{modules}(ms \cup \{\textbf{module}\ M^n = m\})\ \textbf{in}\ E_1.()}$$

$\boxed{P \longrightarrow P'}$ (internal computation step of a program)

$$\text{(ver)}\quad \textbf{modules}\ ms\ \textbf{in}\ E_1.(M^n.z) \longrightarrow \textbf{modules}\ ms\ \textbf{in}\ E_1.v \quad \text{where } ms = \{.\ , \textbf{module}\ M^n = \{.\ z{:}T = v\ .\ \}, .\ \}$$

$$\text{(unver)}\quad \textbf{modules}\ ms\ \textbf{in}\ E_1.(M.z) \longrightarrow \textbf{modules}\ ms\ \textbf{in}\ E_1.v \quad \text{where } ms = \{.\ , \textbf{module}\ M^n = \{.\ z{:}T = v\ .\ \}, .\ \} \text{ and } n = \text{maxversion}(M, ms)$$

$$\text{(let)}\quad E_2.E_1.\textbf{let}\ x{:}T = v\ \textbf{in}\ e \longrightarrow E_2.E_1.\{v/x\}e$$

$$\text{(proj)}\quad E_2.E_1.\pi_r\,(v_1, v_2) \longrightarrow E_2.E_1.v_r$$

$$\text{(app)}\quad E_2.E_1.(\lambda x{:}T.e)v \longrightarrow E_2.E_1.\{v/x\}e$$

where

$$\text{mods}(\{\textbf{module}\ M_1^{n_1} = m_1, .\ , \textbf{module}\ M_k^{n_k} = m_k\}) = \{M_1^{n_1}, .\ , M_k^{n_k}\}$$
$$\text{maxversion}(M, ms) = \max\{n \mid M^n \in \text{mods}(ms)\}$$

Fig. 16. The $\lambda^{\text{mod}}_{\text{update}}$ calculus reduction rules.

to be the initial set of modules, an initial expression $M.f\ ()$, and $m \equiv \{z = (5,5)\}$ be a module body to be loaded. We have:

$$\begin{array}{rl} & \textbf{modules}\ ms\ \textbf{in}\ M.f\ () \\ \longrightarrow & \textbf{modules}\ ms\ \textbf{in}\ (\lambda x{:}\mathsf{unit}.\textbf{let}\ y{:}\mathsf{unit} = \textbf{update}\ \textbf{in}\ M.z)\ () \\ \longrightarrow & \textbf{modules}\ ms\ \textbf{in}\ \textbf{let}\ y{:}\mathsf{unit} = \textbf{update}\ \textbf{in}\ M.z \\ \xrightarrow{M^1=m} & \textbf{modules}\ ms'\ \textbf{in}\ \textbf{let}\ y{:}\mathsf{unit} = ()\ \textbf{in}\ M.z \\ \longrightarrow & \textbf{modules}\ ms'\ \textbf{in}\ M.z \\ \longrightarrow & \textbf{modules}\ ms'\ \textbf{in}\ (5,5) \end{array}$$

where $ms' = ms \cup \{\textbf{module}\ M^1 = \{z = (5,5)\}\}$.

At the point where $M.f$ is resolved, in the first reduction step, the greatest extant version of M is M^0—so $M.f$ is replaced by its $M^0.f$ definition. When $M.z$ is resolved in the last reduction step, however, the greatest version of M is M^1 supplied by the update—and so $M.z$ resolves to $(5,5)$ instead of 3. Notice that the update changes the type of $M.z$ from $\mathsf{int} * \mathsf{int}$ to int. This is acceptable because it does not admit the possibility of any run-time type errors (though it causes the type of the final result to change), as determined by re-type checking the program at update-time (more on this in the next subsection).

Here Γ ranges over partial functions from identifiers x to types T, and Σ ranges over partial functions from module names M and versioned module names M^n to module types σ. Define

$$\begin{array}{l}
\mathrm{modsig}(\{z_1{:}T_1 = v_1 \ldots z_n{:}T_n = v_n\}) \;=\; \{z_1{:}T_1, \ldots, z_n{:}T_n\} \\
\mathrm{modctx0}(\{\mathbf{module}\ M_1^{n_1} = m_1, \ldots, \mathbf{module}\ M_k^{n_k} = m_k\}) \;= \\
\quad \{M_1^{n_1}{:}\mathrm{modsig}(m_1), \ldots, M_k^{n_k}{:}\mathrm{modsig}(m_k)\} \\
\mathrm{modctx}(ms) \;= \\
\quad \mathrm{modctx0}(ms) \cup \{M{:}\sigma \mid \exists\, n.M^n{:}\sigma \in \mathrm{modctx0}(ms) \wedge n = \mathrm{maxversion}(M, ms)\}
\end{array}$$

$\boxed{\varnothing \vdash P{:}T}$ (type checking a program)

$$\frac{\begin{array}{l}\Sigma = \mathrm{modctx}(\{\mathbf{module}\ M_1^{n_1} = m_1, \ldots, \mathbf{module}\ M_k^{n_k} = m_k\}) \\ \Sigma \vdash m_i{:}\mathrm{modsig}(m_i) \quad (i = 1.\ k) \\ \Sigma; \varnothing \vdash e{:}T\end{array}}{\varnothing \vdash \mathbf{modules}\{\mathbf{module}\ M_1^{n_1} = m_1, \ldots, \mathbf{module}\ M_k^{n_k} = m_k\}\ \mathbf{in}\ e{:}T}$$

$\boxed{\Sigma \vdash m{:}\sigma}$ (type checking a module body)

$$\frac{\Sigma; \varnothing \vdash v_i{:}T_i \quad (i = 1.\ n)}{\Sigma \vdash \{z_1{:}T_1 = v_1 \ldots z_n{:}T_n = v_n\}{:}\{z_1{:}T_1, \ldots, z_n{:}T_n\}}$$

$\boxed{\Sigma; \Gamma \vdash e{:}T}$ (type checking an expression)

$$\frac{}{\Sigma; \Gamma \vdash n{:}\mathsf{int}} \qquad \frac{}{\Sigma; \Gamma \vdash ()\mathsf{:unit}} \qquad \frac{\Sigma; \Gamma \vdash e{:}T \qquad \Sigma; \Gamma \vdash e'{:}T'}{\Sigma, \Gamma \vdash (e, e'){:}T * T'}$$

$$\frac{\Sigma; \Gamma \vdash e{:}T_1 * T_2}{\Sigma; \Gamma \vdash \pi_r\ e{:}T_r} \qquad \frac{\Sigma; \Gamma \vdash e{:}T \to T' \qquad \Sigma; \Gamma \vdash e{:}T}{\Sigma; \Gamma \vdash e\ e'{:}T'}$$

$$\frac{\Sigma; \Gamma, x{:}T \vdash e{:}T'}{\Sigma; \Gamma \vdash \lambda x{:}T.e{:}T \to T'} \qquad \frac{\begin{array}{c}\Sigma; \Gamma, x{:}T \vdash e'{:}T' \\ \Sigma; \Gamma \vdash e{:}T\end{array}}{\Sigma; \Gamma \vdash \mathbf{let}\ x{:}T = e\ \mathbf{in}\ e'{:}T'}$$

$$\frac{}{\Sigma; \Gamma, x{:}T \vdash x{:}T} \qquad \frac{}{\Sigma, M^n{:}\{\ldots, z{:}T, \ldots\}; \Gamma \vdash M^n.z{:}T}$$

$$\frac{}{\Sigma, M{:}\{\ldots, z{:}T, \ldots\}; \Gamma \vdash M.z{:}T} \qquad \frac{}{\Sigma; \Gamma \vdash \mathbf{update}\mathsf{:unit}}$$

Fig. 17. The $\lambda^{\mathrm{mod}}_{\mathrm{update}}$ calculus typing rules.

The type system provides the necessary checks to ensure that loading a module does not result in a program that will reduce to a stuck state (one in which the expression is not a value and yet no reduction rule applies). Figure 17 shows the type system for our calculus. The rules for the judgement $\Sigma; \Gamma \vdash e{:}T$ are the standard ones for the simply typed λ calculus, extended in the obvious way to deal with the typing of module components. As in $\lambda_{\mathrm{update}}$, the **update** command is statically uninteresting and types as unit, as this is the type of the () value it becomes after (update). The other two judgements are more interesting. The judgement $\Sigma \vdash P{:}U$

types whole programs and handles most of the complexity in typing modules. We use two auxiliary functions modsig and modctx: modsig determines the interface of a module given its body and, given a set of modules, modctx determines the partial function that maps versioned module names M^n to their signatures and also maps the unversioned module names M to the signature of the highest-versioned module with the same name. The function modctx can thus be used to determine the module context in which the program (including the module bodies themselves) should be typed. The single rule defining the judgement ensures that the expression and every module body can be typed in this context; this means that the modules are allowed to be mutually recursive, as every module name is available in the typing of each module.

Typing of module bodies is expressed by the judgement $\Sigma \vdash m : \sigma$, that is, that module body m has interface σ in the context of module declarations Σ; it simply requires that each component of the module has the appropriate type.

7.3 Discussion

Many design decisions reflect our aim to keep the $\lambda^{\text{mod}}_{\text{update}}$ calculus simple, but nonetheless, practical and able to express different updating strategies for programs. We further consider some of those design decisions here.

The calculus addresses the run-time mechanisms involved in implementing updating (i.e. loading new modules and allowing existing code or parts thereof to refer to them), but does not cover all the important software development issues of managing updateable code. In practice, we would expect compiler support for aiding the development process (Hicks 2001). For example, user programs could refer to the 'current' and 'previous' versions of a module, and the compiler would fill in the absolute version number.

An important design question of language-level updating systems is whether an old version and a new version of a binding may coexist. While some systems prefer one version at a time (Gupta 1994; Gilmore *et al.* 1997), many systems allow multiple versions to coexist, possibly indefinitely (Frieder & Segal 1991; Armstrong *et al.* 1996; Peterson *et al.* 1997; Duggan 2001; Hicks 2001).[5] The λ_{update} calculus ensures that each 'use' of an identifier (e.g. a function call) will be the most recent version. Thus, it is possible that an old and a new version will be active if the old version was active at the time of the dynamic update. At best, this coexistence is indirect (with the intention that the older versions will eventually become redundant and then thrown away). By contrast, the use of module versions in $\lambda^{\text{mod}}_{\text{update}}$ allows multiple generations of a module to exist simultaneously, and provides explicit control over which version of a module we are referring to, allowing us to delimit the effect of an update.

We chose to use λ_r-style reduction in $\lambda^{\text{mod}}_{\text{update}}$ to keep things simple. We could have used λ_d in the obvious way—that is, by making expressions $M.z$ (but not $M^n.z$) into values, and by partitioning evaluation contexts to include destruct contexts.

[5] A. Appel, unpublished data, December 1994.

Some of the consequences of choosing λ_r rather than λ_d were discussed at the end of the last section. Interestingly, the semantics of Erlang, upon which $\lambda^{\text{mod}}_{\text{update}}$ was based, is silent on the meaning of higher-order functions and updates. The informal specification (Barklund & Virding 1999) states that a direct external call (of form $M.z\ e_1 \ldots e_n$ in $\lambda^{\text{mod}}_{\text{update}}$) should be to the newest version of module M. However, there is nothing said of what would happen should $M.z$ be passed as a functional argument that is ultimately applied after its definition has been updated. The current version of Erlang at the time of this writing (5.5) uses λ_d semantics in this case, as with λ_{update}. Oddly, versioned calls (like $M^n.z$ in $\lambda^{\text{mod}}_{\text{update}}$) are also given this semantics—they will reduce to the updated version! We view this incongruity as evidence of the need to formally define the meaning of updates. (We note that the most recent 'core Erlang' formal specification, Carlsson *et al.* 2004, is silent on the semantics of updates.)

Finally, similarly to λ_{update}, module updates in $\lambda^{\text{mod}}_{\text{update}}$ must leave the program well typed following the update. However, in contrast to λ_{update}, an update to a module is permitted to change the types of that module's member elements. This allows updates to be more flexible (indeed, it is critical in practice; Neamtiu *et al.* 2006), but imposes a greater burden at link-time: the entire program must be checked following the update, rather than just the updated element in the context in which it appears. To see why this is necessary, consider a reference to $M.z$. When the program was initially checked, z might have type T but in the new version it could have type T'. References to $M.z$ could appear in the active computation (i.e. the e part in **modules** ms **in** e) and the existing modules. We must type check the whole program to ensure that any such references are still correctly typed.

However, rather than rechecking the entire program at link-time, we could imagine 'precomputing' the relevant information about the existing code, and then using that information to preclude problematic updates. We have explored a type system for supporting such an approach in related work on a language called Proteus (Stoyle *et al.* 2005). Although we have not made use of this type system here, to keep things simpler, we briefly sketch how it would work. Adapted to $\lambda^{\text{mod}}_{\text{update}}$, a Proteus-style type system would identify, for each **update** point, those module-level identifiers $M.z_1 \ldots M.z_n$ the program could eventually reference from non-updated code after the update; if those identifiers change type, we would have a type violation. Such non-updated code could either be in the active expression or be in modules called from the active expression through versioned calls $M^n.f$ (since such functions could make 'outdated' unversioned calls). Whenever an **update** point is reached, a dynamic update is only permitted to take effect if the loaded code is itself well typed (as the condition required by the λ_{update} updating rule), and if no identifier in $M.z_1 \ldots M.z_n$ changes type.

7.4 Example: Updating a server application

To illustrate the expressive power of our calculus, we present more realistic examples of updating a long-lived server application. There are many real-world examples of this class of system that employ or could benefit from DSU; for example,

```
modules {
  module Handlers[1] = {
    handleGet = λ(q,e). ...
    handlePost = λ(q,e). ...
    handleUpdate = λ(q,e). update ; q
    ...
    }
  module Server[1] = {
    getevent = λq. ...
    handle = λq.
      let (q,e) = Queue.dequeue q in
      ...demultiplex...
      ... Handlers[1].handleGet (q,e) ...
      ... Handlers[1].handlePost (q,e) ...
      ... Handlers[1].handleUpdate (q,e) ...
    loop = λq.
      let q = Server.getevent q in
      let q = Server.handle q in
      Server.loop q
    }
  } in
Server.loop Queue.empty
```

Fig. 18. An updateable (web) server.

financial transaction processors, web and database servers, network routers, intrusion detection sensors, and more. Because our calculus lacks concurrency (a non-trivial extension), we focus on a single-threaded, event-based architecture, which is not uncommon in server applications (Pai *et al.* 1999; Welsh *et al.* 2001; Boa n.d.).

To make the code examples easier to read, we have taken some liberties with our syntax. In particular, we allow tuples rather than pairs, with pattern-matching; we have elided all typing annotations; we assume the existence of booleans and conditionals; we assume the existence of a type of queues, which could be implemented using lists; and we allow simultaneous updates of multiple modules. All these should be clear in context and would be routine to add to the calculus definition.

7.4.1 Initial system

The initial program for our updateable server is shown in Figure 18. The `Server` module implements the basic event loop. The `loop` function has a queue of events, for example, HTTP requests from clients or responses sent by handlers. New events are created by `getevent`, which queues any new events (such as client requests) and returns, or blocks if both the queue is empty and no new events have occurred. Once an event is extracted, it is demultiplexed by the `handle` function, which calls the handlers implemented in the `Handlers` module. One possible event is a request to update the server. This is processed by the `Handlers.handleUpdate` function,

```
module Log[1] = {
  emptyLog = ...
  logevent = λ(l,e). ...
  }
module Server[2] = {
  getevent = λq. ...
  handle = λ(l,q).
    let (q,e) = Queue.dequeue q in
    let l = Log.logevent (l,e) in
    ...
    Handlers[1].handleGet (q,e)
    ...
  loop = λq.
    Server[2].loop' (Log[1].emptyLog, q)
  loop' = λ(l,q).
    let q     = Server.getevent q in
    let (l,q) = Server.handle (l,q) in
    Server.loop' (l,q)
  }
```

Fig. 19. Adding a log to the server.

which invokes **update**. Notice that because of the placement of the **update** primitive, updates will always occur just before the recursive call to `loop`, meaning that no computations will be incomplete when the update is accepted. This intuitively allows us to believe that an update will not result in an inconsistent state. Also notice that calls to `Handlers` functions use an explicit version; the reason for this will be evident shortly. We assume that the program is using a module `Queue` to implement its event queue, which is not shown. The programs starts with a call to `loop` with an empty queue as its argument.

7.4.2 *First update: adding a log*

As a first example update, say we want to log all of the HTTP events that we process, so we want to add logging to the server. To do this, we need to change the `loop` and `handle` functions of `Server` to additionally take a log object as an argument. Upon handling each event, a record will be added to the log. To realise this change online, we note that the existing `Server.loop` function does not expect this extra parameter, and therefore we must introduce a 'transitional' function `loop` *at the old type* (i.e. expecting only a queue as its argument), which then calls the new version of `loop'` at the new type (i.e. expecting both the queue and the log as arguments). This is shown in Figure 19. Transitional functions have been proposed in existing systems under various names and guises (Lee 1983; Frieder & Segal 1991; Hicks 2001). Note that we do not need to add a transitional function for `handle` since, following the update, it can only ever be called from `loop'` (at the new type).

When the original `Server`^{1}`.loop` function handles the update event, its recursive call to `Server.loop` will dispatch to the new `Server`^{2}`.loop` function by the (unver) reduction rule. This function creates an empty log and calls `Server`^{2}`.loop'`, which continues processing events. This function will call the new `handle` function, which expects the log as an argument, and will log the appropriate events.

It would seem unfortunate from a software engineering point of view that the name of the `loop` function changes to `loop'` in the new version. However, this issue can be resolved with compiler/tool support, as we have motivated and implemented in other work (Hicks 2001; Neamtiu *et al.* 2006). Our goal is to focus on the run-time issues, since doing so keeps things simpler and will not impede our formal reasoning ability.

7.4.3 Second update: Enriching events

The first update added to the server's functionality; we might also wish to enrich or change existing functionality. For instance, we could extend the definition of events to include additional information, perhaps to refine existing event descriptions or to add new ones. Such a change will impact all of the code that manipulates events, including our event queue and handler functions.

Because we are making the change without shutting down the system, we have to consider the existing unprocessed events before switching to the new format. In particular, the server's existing event queue could be nonempty. Here are two possible choices: (1) convert all of the events in the existing queue to have the format expected by the new code, or (2) process the old events using the old code and then switch to using new code for the new events. The former strategy, which we shall dub *convert*, is taken by most proposed DSU systems (e.g. Lee 1983; Frieder & Segal 1991; Gilmore *et al.* 1997; Duggan 2001; Hicks 2001; Stoyle *et al.* 2005; Neamtiu *et al.* 2006; Squeak n.d.) while the latter, which we dub *complete*, is taken by fewer systems (e.g. Appel 1994; Peterson *et al.* 1997). Each approach has advantages and disadvantages; a main goal for our calculus is to be able to express a range of design decisions, so as to evaluate these trade-offs.

In both cases, we will need to create new versions of the `Handlers` and `Log` modules that use the new form of events:

```
module Handlers2 = ...handles new event type
module Log2 = ...handles new event type
```

(we omit the details of these two).

The new `Server` module for the *complete* strategy is illustrated in Figure 20. Here, if the existing queue is nonempty, the new `loop'` processes the old events by explicitly referring to `Server`^{2}`.getevent` and `Server`^{2}`.handle` functions, which call the handlers from the `Handlers`^{1} module. Once the queue is empty, `loop'` calls the new `Server`^{3}`.loop''` function with an empty queue (to hold the new events).

Note that the *complete* strategy would not have worked if we had not explicitly included the version number when calling `Handlers`^{1} handling functions in `Server`^{2}`.handlers`. If instead we had used the unversioned syntax (e.g.

```
module Server^3 = {
  getevent = λq. ...an event of the new type...
  handle = λ(l,q).
    let (q,e) = Queue.dequeue q in
    let l = Log.logevent (l,e) in
    ...
    Handlers^2.handleGet (q,e)
    ...
  loop' = λ(l,q).
    if (Queue.isempty q) then
      Server^3.loop'' (l, Queue.empty)
    else
      let q = Server^2.handle (l,q) in
      Server^3.loop' (l,q)
  loop'' = λ(l,q).
    let q     = Server.getevent q in
    let (l,q) = Server.handle (l,q) in
    Server.loop'' (l,q)
  }
```

Fig. 20. Complete existing events first.

`Handlers.handleGet`), the new versions of the handlers would have been called following the update, and this would have been flagged as a type error. On the other hand, had we used the unversioned syntax, we could have supplied an intermediate update that inserted the versioned variable syntax, and then proceeded with our original update!

The new module required for the *convert* strategy is shown in Figure 21. Here, we have defined two new functions `convertevent` and `convert`; the former converts an event from the old to the new format, and the latter recursively creates a new queue containing the converted events of the old one. At the time of the update, the `loop'` function will call `convert` to create a new queue, and then call `Server`^{3}`.loop''` to proceed with processing the converted (and new) events with the new code.

8 Related work

8.1 Lambda calculi

As discussed in §3.2, 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 (Abadi *et al.* 1990) and on sharing in call-by-need languages (Ariola *et al.* 1995).

In work on the compilation of extended recursion (particularly for mixin modules) Hirschowitz *et al.* have (independently) used a semantics that is similar to λ_d, save that (a) the language allows more general recursive definitions and (b) the semantics collapses multiple **let**s (Hirschowitz 2003; Hirschowitz *et al.* 2003). It draws on the work of Ariola and Blom (2002), which also collapses **let** blocks. For rebinding, we need to preserve this structure.

```
module Server^3 = {
  convertevent = λe. ...convert event e
  convert = λ(q,q').
    if (Queue.isempty q) then q'
    else
      let (q,e) = Queue.dequeue q in
      let e' = Server^3.convertevent e in
      let q' = Queue.enqueue q' e' in
      Server^3.convert (q,q') in
  getevent = λq. ...an event of the new type
  handle = λ(l,q).
    let (q,e) = Queue.dequeue q
    let l = Log.logevent (l,e)
    ...
    Handlers^2.handleGet (q,e)
    ...
  loop' = λ(l,q).
    let q' = Server.convert (q,Queue.empty) in
    Server^3.loop'' (l,q')
  loop'' = λ(l,q).
    let q     = Server.getevent q in
    let (l,q) = Server.handle (l,q);
    Server.loop'' (l,q)
  }
```

Fig. 21. Convert existing events to new format.

There are also similarities with Felleisen and Hieb's (1992) syntactic theory of state. 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. As in our system, there is a notion of a 'final answer', which may require further clean-up to yield the value that is the result of the computation in the usual calculus (our $[\![.]\!]$ function).

8.2 Dynamic rebinding and λ_{marsh}

Dynamic binding

Work on dynamic binding can be roughly classified along three dimensions. Firstly, 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. Secondly, one can work either with one class of variables or split into two: one treated statically and one dynamically. Thirdly, 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 (1998) and Vivas Frontana (2001) for further discussion.

Dynamic scoping first appeared as a bug in McCarthy's Lisp 1.0, and has survived in most modern Lisp dialects in some form. There it is usually referred to as 'dynamic binding'. Lisp 1.0 had one class of variables. MIT Scheme's (MIT n.d.) `fluid-let` form and Perl's `local` declaration similarly perform dynamically scoped rebinding of variables. Modern Lisp distinguishes between dynamically and statically scoped variables at declaration time, as formalised in the λ_d calculus of Moreau (1998). Lewis *et al.* (2000) propose to add syntactically distinct, dynamically scoped *implicit parameters* 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 (1993) and the λN calculus of Dami (1998) both have two classes of variable, with a rebinding primitive that specifies new bindings for individual variables. Jagannathan's (1994) *Rascal* language 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 (2001) introduce a typed context calculus for expressing first-class evaluation contexts within the λ calculus. Context holes can be 'filled in' with terms having free variables that are captured by the surrounding context. This allows binding at context-application time, but does not support rebinding. This is developed in the *MobileML* language (Hashimoto & Yonezawa 2000). Garrigue (1995) 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, for example, for repeatedly mobile code. As argued in §4, 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 (1987) 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*: **mark** marks a *binding* context, whereas # 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 (Felleisen *et al.* 1988), as used by Queinnec (1993), 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.* (1995) have studied # and $\mathscr{F}$ 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. §4.4). Very recently, Kiselyov *et al.* (2006) have studied the problems of combining dynamic binding with delimited control (such as the control operators of Gunter *et al.*). They show how these two features, when combined, result in a system with a number of undesirable features. They propose expressing dynamically bound parameters in terms of delimited control prompts. It would be interesting future work to examine in detail similar translations of λ_r and λ_d.

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 (Queinnec 1993; Moreau & Queinnec 1994).

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. Conventional executables will, when run, dynamically link shared libraries for standard library functions (e.g. `read`, `write`). Which libraries are loaded depends upon the context; for example, a machine might have a library compiled with profiling enabled and one without. However, once dynamically bound, a variable's definition is fixed, precluding rebinding for marshalling or update. Modern languages often provide an interface to the dynamic linker so that programs can load new code at run-time (Armstrong *et al.* 1996; Rouaix 1996; Leroy *et al.* 2001; Drossopoulou & Eisenbach 2002; dlopen n.d.). Dynamic linking has been formally modelled for low-level machine code (Duggan 2000; Hicks *et al.* 2000; Hicks & Weirich 2000), and high-level languages such as Java (Drossopoulou & Eisenbach 2002). Several authors have considered customised linking for security, performance or debugging purposes (Rouaix 1996; Hicks *et al.* 2000; Serra *et al.* 2000; Sewell & Vitek 2000).

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 (Cardelli & Gordon 1998; Riely & Hennessy 1999; Sewell

et al. 1999; Chothia & Stark 2000; Sewell & Vitek 2000; Schmitt 2002). 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 (Fournet *et al.* 1996; Sewell *et al.* 1999) 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 (Schmitt 2002).

8.3 Dynamic update

There has been steady interest in how to dynamically update running systems since at least Fabry (1976). Frieder and Segal (1991), Hicks (2001), and Ajmani (2004) each has surveyed (overlapping) portions of the literature. Recent work has explored dynamic updating for operating systems (Soules *et al.* 2003; Baumann *et al.* 2004, 2005; Potter & Nieh 2005; Chen *et al.* 2006), servers (Altekar *et al.* 2005; Stoyle *et al.* 2005; Neamtiu *et al.* 2006), distributed systems (Ajmani *et al.* 2006) and persistent object stores (Boyapati *et al.* 2003), and many languages provide some form of support for DSU, including Erlang (Armstrong *et al.* 1996), Smalltalk (Goldberg & Robson 1989) and Java (Java n.d.). As mentioned earlier, while Erlang has a core semantics (Carlsson *et al.* 2004), the semantics does not consider dynamic updating. Indeed, most past work has focused on implementation issues; our current work focuses on defining rigorous semantics. There is a small collection of related work in this area.

Duggan (2001) defines a formal language in which module types may be converted lazily during program execution, rather than at once during garbage collection. As a result, different versions of a type/module may coexist during program execution, and must be convertible from the old to new version and *vice versa*. A novel type system is presented and type soundness is proved. In this system, code updating is treated informally, based on arguments around reference types. The assumption is that a compiler will introduce an extra level of indirection, converting all updateable definitions of type T be ones of type T **ref**. While expedient, such informality side steps the question of higher-level semantics: following an update, which program elements should 'notice' the change? In other words, one must decide, when writing Duggan's hypothetical compiler, where the dereference operations should be inserted. Our λ_r and λ_d reduction strategies provide two possible answers: in redex position or in destruct position. Moreover, we have shown that these two strategies are 'essentially call-by-value' (Theorem 4), corresponding the original (precompilation) semantic view of the program.

Dynamic ML (Gilmore *et al.* 1997) is a proposed implementation of ML with a formalised abstract machine (Walton 2001) that enables replacement of modules at run-time; changes can include the alteration of abstract types and the addition (and possibly deletion) of module definitions. To ensure soundness, existing values of changed abstract types will be converted to the new representation during a garbage-collection phase at update-time. Different versions of modules may not

coexist, and a module must be inactive (e.g. not on the run-time stack) before it can be replaced.

Both Duggan's formalism and Dynamic ML focus primarily on the problem of converting values of changed types following an update, adopting particular language mechanisms to do so. We believe the wider questions of how and when to ensure safe updates require attention; a prime goal of the λ_{update} and $\lambda^{\text{mod}}_{\text{update}}$calculi has been to use the simplest mechanisms possible in order to highlight commonality among various DSU systems. The fact that Erlang and its update mechanisms are in wide use strongly motivated it as the inspiration for $\lambda^{\text{mod}}_{\text{update}}$; to our knowledge neither Dynamic ML nor Duggan's system has been implemented.

Using the foundations presented here, we recently formalised a calculus called Proteus (Stoyle *et al.* 2005) for modelling updates in imperative programs and implemented it in a C compiler called Ginseng (Neamtiu *et al.* 2006). In Proteus, updates occur at the granularity of individual definitions (whether types, variables or functions) as in λ_{update} (rather than modules as in $\lambda^{\text{mod}}_{\text{update}}$). Replacement definitions may have different types in comparision with their originals, and a novel type system based on *capabilities* (Walker *et al.* 2000) is used to ensure type safety. The fact that possible update points are made explicit in the program text is a key to balancing the flexibility and safety of the system.

The implementation in Ginseng has been shown to scale to realistic programs. In particular, we dynamically updated three open-source programs: the "Very Secure" FTP daemon, `vsftpd`, OpenSSH's `sshd` daemon and the `zebra` routing daemon from the GNU Zebra routing package. These programs range in size from 10 to 58 KLOC. We were able to start program versions at least 3 years old and then subsequently update them, while they ran, with code implementing each subsequent release. This resulted in as many as 12 successful dynamic updates.

In summary, Proteus and Ginseng benefit from the basic insights presented here and demonstrate that our core rebinding ideas can be scaled to practical implementations.

9 Conclusions

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;
- demonstrated that dynamic update of functional programs can be expressed using similar mechanisms by introducing the λ_{update} calculus—again with a simple destruct-time semantics; and

- showed how the basic ideas in λ_{update} can be scaled up to a more realistic calculus $\lambda^{\text{mod}}_{\text{update}}$, which permits updates at the module level. This calculus models a more general form of the updating capability in the functional programming language Erlang (Armstrong *et al.* 1996).

These ideas have been used to build realistic programming languages for distributed programming, such as Acute (Sewell *et al.* 2004, 2007) and HashCaml (Billings *et al.* 2006), and dynamic updating, such as Proteus (Stoyle *et al.* 2005) and Ginseng (Neamtiu *et al.* 2006). These languages are under active development, with ongoing research on language designs that harmonise the tensions of flexibility, efficiency and safety for realistic programs. We refer the reader to our papers on these languages for specific discussion of various possible future directions.

Acknowledgments

We acknowledge support from a Royal Society University Research Fellowship (Sewell), a Marconi EPSRC CASE Studentship (Stoyle), a St. Catharine's College Heller Research Fellowship (Wansbrough), EPSRC grants GRN24872 and GRT11715, 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.

Appendix

Observational equivalence between λ_r, λ_c and λ_d

In this appendix, we outline our proof that λ calculus with our delayed instantiation reduction strategies has the same observational equivalence as with the standard CBV reduction strategy.

To show these results, we build a particular form of (weak) bisimulation that we call eventually weak (bi)simulation (EWS/EWB).

Definition 3 (*Eventually weak (bi)simulation*)
Given two transition systems $X \subseteq S_1 \times S_1$ and $Y \subseteq S_2 \times S_2$, we say that a relation $R \subseteq S_1 \times S_2$ regarding states of X to states of Y is an *eventually weak simulation from X to Y* if and only if for every $e_x\ R\ e_y$ the following holds:

$$e_x \longrightarrow_X e'_x \implies \exists\, e'_y.\ e_y \longrightarrow^*_Y e'_y \wedge \exists\, n \geqslant 0.\ e'_x \rightarrow^n_X e''_x \wedge e''_x\ R\ e'_y$$

If the implication holds after interchanging X and Y, then R is said to be an *eventually weak bisimulation between X and Y*.

It is straightforward to see that every weak bisimulation is an eventually weak bisimulation, and hence that every bisimulation is an eventually weak bisimulation.

Informally, we require the weakness relaxation because λ_r performs more work than λ_c: while λ_c instantiates all instances of a bound variable in a single reduction step, λ_r requires reductions proportional to the number of occurrences of the variable. In addition, we require the eventually weak relaxation as λ_c's recursive

function expansion results in occurrences of $\lambda x.\mathbf{letrec}\ z = \lambda x.e\ \mathbf{in}\ e$ while in λ_r, we obtain occurrences of $\lambda x.e$ instead. For example:

$$\begin{array}{lll} \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ z\ n & \longrightarrow_c & (\lambda x{:}T.\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ e)n \quad \text{(letrec)} \\ \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ z\ n & \longrightarrow_r & \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ (\lambda x{:}T.e)n \quad \text{(instrec)} \end{array}$$

Upon application of these functions, λ_c requires an extra reduction when compared with λ_r, that is:

$$\begin{array}{lll} (\lambda x.\mathbf{letrec}\ z = \lambda x.e\ \mathbf{in}\ e)v & \longrightarrow_c & \mathbf{letrec}\ z = \lambda x.e\ \mathbf{in}\ \{v/x\}e \\ & \longrightarrow_c & \{\lambda x.\mathbf{letrec}\ z = \lambda x.e\ \mathbf{in}\ e/z\}\{v/x\}e \end{array}$$

but

$$(\lambda x.e)u \quad \longrightarrow_r \quad \mathbf{let}\ x = u\ \mathbf{in}\ e$$

A similar situation holds for λ_d.

The proofs of observational equivalence between λ_r and λ_c as well as λ_d and λ_c are similar. In the interests of space, we give only details of the equivalence between λ_r and λ_c. Details of the other proof can be found in Stoyle's PhD thesis (Stoyle 2006).

A.1 $\lambda_r \leftrightarrow \lambda_c$ observational equivalence

First, we restate the theorem we intend to prove in this section.

Theorem 7
For all $e \in \lambda$, the following hold:

1. $\vdash e{:}\mathsf{int} \implies (e \longrightarrow_c^* n \implies \exists u.\, e \longrightarrow_r^* u \wedge n = [\![u]\!])$
2. $\vdash e{:}\mathsf{int} \implies (e \longrightarrow_r^* u \implies \exists n.\, e \longrightarrow_c^* n \wedge n = [\![u]\!])$

Rather than explicitly construct an eventually weak bisimulation relation between λ_r and λ_c terms (which seems to be very difficult), we actually first introduce an annotated calculus, $\lambda_{r'}$, that records more information during reduction. We can then show observational equivalence between λ_r and λ_c by defining an EWB between λ_r and $\lambda_{r'}$, and showing that the termination relation for $\lambda_{r'}$ and λ_c coincide.

To motivate the intermediate calculus $\lambda_{r'}$, we observe that not every value-binding let in a λ_r term is part of 'the environment'. Let-bindings on the outside of a computation that bind values are morally part of the computation's environment; their values are used by the computation, but the terms they bind are fully computed. For example, in

$$\mathbf{let}\ x = 5\ \mathbf{in}\ \mathbf{let}\ y = 6\ \mathbf{in}\ \mathbf{let}\ z = \pi_1(x, y)\ \mathbf{in}\ z$$

the values bound to x and y are used by the computation under them, but no more computation occurs above or within them. The variables x and y are part of the environment, but the **let** binding z is in the part of the program that is yet to be computed: it is part of the 'computation'.

Integers	n		
Identifiers	x, y, z		
Types	T	::=	$\mathsf{int} \mid \mathsf{unit} \mid T * T' \mid T \to T'$
Exprs	a	::=	$z \mid n \mid () \mid (a, a') \mid \pi_r\ a \mid \lambda^j x.a$
		$\mid$	$a\ a' \mid \mathbf{let}_m\ z = a\ \mathbf{in}\ a' \mid \mathbf{letrec}_m\ z = \lambda x{:}T.a\ \mathbf{in}\ a'$
Annotations	m	::=	$0 \mid 1$
	j	::=	$\cdot \mid z$
	r	::=	$1 \mid 2$

Fig. A 1. Annotated syntax for $\lambda_{r'}$.

The intermediate language, $\lambda_{r'}$, given in Figures A 1 and A 2, explicitly distinguishes these two forms of **let** . Zero-tagged lets (**let**$_0$) for environment-**let** s and one-tagged lets (**let**$_1$) for program-**let** s. The (zero) and (zerorec) reductions convert a one-tagged **let** /**letrec** into a zero-tagged **let** /**letrec** whenever a one-tagged **let** /**letrec** binding a value is in redex position. These reductions correspond to substituting **let** s away in λ_c.

Similarly, a tagging scheme is employed for distinguishing between functions and recursive unrollings of functions. Whenever a variable bound by a **letrec** is instantiated, we tag the function with the name of the **letrec** it came from, for example, z will be instantiated to $\lambda^z x.a$. See the (instrec) rule in Figure A 2.

Notation We say that $\lambda^z x.a$ is a *recursive function* and call z in that term a *recursive variable*. Write frv(a) (the *free recursive variables* in a) for the recursive variables in a not bound by an enclosing **letrec** and frf(a) (the *free recursive functions* in a) for the recursive functions whose recursive variables are in frv(a).

Notation Whenever we want to specify that a reduction is of a specific type, we will label the transition with its name, for example, $a \xrightarrow{\text{inst}}_{r'} a'$. As a generalisation of this, we will write $a \xrightarrow{\text{insts}}_{r'} a'$ to mean that a can do an inst or instrec transition to become a', and we call the action an *inst reduction*. Similarly, we write $a \xrightarrow{\text{zeros}}_{r'} a'$ for a zero or zerorec transition and call it a *zero reduction*.

Notation We write $\mu(z, x, T, e)$ for $\lambda x{:}T.\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ e$ and overload it for annotated terms such that $\mu(z, x, T, a)$ stands for $\lambda x{:}T.\mathbf{letrec}_1\ z = \lambda x{:}T.a\ \mathbf{in}\ a$. Whenever the types are clear from the context, we write $\mu(z, x, e)$ and $\mu(z, x, a)$, respectively.

Intuitively, environment-**let** s are found only on the outside of a computation, and subsequent to a program-**let** occurring there should be no more occurrences of environment-**let** s. Also, we expect tagged functions to be introduced only by the computation and not by the user, therefore we do not expect to find tagged functions under λ's or below program **let** s. We call such terms *well-formed*, and write wf[a] to denote that a term a is well formed. For reasons of space, we suppress the rather obvious definition.

In Figure A 3, we define a function $[\![-]\!]^{\Phi}$ that translates a $\lambda_{r'}$ term into an 'equivalent' λ_c term. The parameter Φ is an *environment*, which is essentially a record of (value) substitutions. It is defined as follows.

Reduction contexts

Values	u	::=	$n \mid () \mid (u, u') \mid \lambda x{:}T.a \mid \mathbf{let}_0\ z{:}T = u\ \mathbf{in}\ u$
		$\mid$	$\mathbf{letrec}_0\ z{:}T = \lambda x{:}T.a\ \mathbf{in}\ u$
Atomic eval ctxts	A_1	::=	$(\_, a) \mid (u, \_) \mid \pi_r\ \_ \mid \_\ a \mid \lambda x{:}T.a\ \_$
		$\mid$	$\mathbf{let}_1\ z{:}T = \_\ \mathbf{in}\ a$
Atomic bind ctxs	A_2	::=	$\mathbf{let}_0\ z{:}T = u\ \mathbf{in}\ \_ \mid \mathbf{letrec}_0\ z = \lambda x{:}T.a\ \mathbf{in}\ \_$
Eval ctxts	E_1	::=	$\_ \mid E_1.A_1$
Bind ctxts	E_2	::=	$\_ \mid E_2.A_2$
Reduction ctxts	E_3	::=	$\_ \mid E_3.A_1 \mid E_3.A_2$

Reduction rules

(proj)	$\pi_r\ (E_2.(u_1, u_2))$	$\longrightarrow$	$E_2.u_r$
(app)	$(E_2.(\lambda x{:}T.a)u)$ if $\mathrm{fv}(u) \notin \mathrm{hb}(E_2)$	$\longrightarrow$	$E_2.\mathbf{let}_0\ x = u\ \mathbf{in}\ a$
(inst)	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.z$ if $z\ \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(u) \notin\ z, \mathrm{hb}(E_3)$	$\longrightarrow$	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.u$
(instrec)	$\mathbf{letrec}_0\ z = \lambda x{:}T.a\ \mathbf{in}\ E_3.z$ if $z\ \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(\lambda x{:}T.a) \notin\ z, \mathrm{hb}(E_3)$	$\longrightarrow$	$\mathbf{letrec}_0\ z = \lambda x{:}T.a\ \mathbf{in}\ E_3.\lambda^z x{:}T.a$
(zero)	$\mathbf{let}_1\ z = u\ \mathbf{in}\ a$	$\longrightarrow$	$\mathbf{let}_0\ z = u\ \mathbf{in}\ a$
(zerorec)	$\mathbf{letrec}_1\ z = \lambda x{:}T.a\ \mathbf{in}\ a'$	$\longrightarrow$	$\mathbf{letrec}_0\ z = \lambda x{:}T.a\ \mathbf{in}\ a'$
(cong)	$\dfrac{a \longrightarrow a'}{E_3.a \longrightarrow E_3.a'}$		

Fig. A 2. $\lambda_{r'}$ calculus.

Definition 4 (*Environment*)
An *environment* Φ is an ordered list containing pairs whose first component is an identifier and whose second component is a c-value or an identifier. An environment is well formed if the following hold:

(i) Whenever $(x, z) \in \Phi$, then $x = z$.
(ii) Whenever $(x, e) \in \Phi$, then for all $z\ \in \mathrm{fv}(e)$ it holds that $z \leqslant_\Phi x$ where $\leqslant_\Phi$ is the ordering of the identifiers in Φ.
(iii) All of the first components of the pairs in the list are distinct.

When Φ is well formed, we write $\Phi \blacktriangleleft$.[6] We write $\Phi, z \mapsto v$ for the disjoint extension of Φ forming a new environment and $\Phi[z \mapsto v]$ for the environment acting as Φ, but mapping z to v.

Note that in the above definition if Φ is well formed, it does not necessarily follow that the extensions are well formed. Clause (i) in the definition is a simplifying assumption reflecting the fact that if an environment maps an identifier to a nonvalue, then it maps it to itself. Clause (ii) is a closure property ensuring that variables that

[6] We adopt this strange notation as later we extend environment well formedness to environment well formedness w.r.t. a term a, which we write $\Phi \blacktriangleleft a$.

Here we introduce a function that expresses the correspondence between well-formed $\lambda_{r'}$ terms built through instantiation and λ terms built through substitution (in the sense of λ_c reduction). $[\![\, a \,]\!]^{\Phi}$ is a function mapping a $\lambda_{r'}$ expression a and an environment Φ to a λ_c expression. We note that in each of the cases where we extend the environment to associate an identifier with a value, we can ensure that the identifier is fresh for the environment by α-conversion.

$$
\begin{array}{rcl}
[\![\, z \,]\!]^{\Phi} & \triangleq & \Phi(z) \\
[\![\, n \,]\!]^{\Phi} & \triangleq & n \\
[\![\, () \,]\!]^{\Phi} & \triangleq & () \\
[\![\, (a, a') \,]\!]^{\Phi} & \triangleq & ([\![\, a \,]\!]^{\Phi}, [\![\, a' \,]\!]^{\Phi}) \\
[\![\, \pi_r \;\; a \,]\!]^{\Phi} & \triangleq & \pi_r \, [\![\, a \,]\!]^{\Phi} \\
[\![\, \lambda x{:}T.a \,]\!]^{\Phi} & \triangleq & \lambda x{:}T.[\![\, a \,]\!]^{\Phi,\, x \mapsto x} \qquad x \notin \mathrm{dom}(\Phi) \\
[\![\, \lambda^z x{:}T.a \,]\!]^{\Phi} & \triangleq & \Phi(z) \\
[\![\, a \; a' \,]\!]^{\Phi} & \triangleq & [\![\, a \,]\!]^{\Phi} \, [\![\, a' \,]\!]^{\Phi} \\
[\![\, \mathbf{let}_0 \;\; z = a \;\; \mathbf{in} \;\; a' \,]\!]^{\Phi} & \triangleq & [\![\, a' \,]\!]^{\Phi,\, z \mapsto [\![\, a \,]\!]^{\Phi}} \qquad z \notin \mathrm{dom}(\Phi) \\
[\![\, \mathbf{let}_1 \;\; z = a \;\; \mathbf{in} \;\; a' \,]\!]^{\Phi} & \triangleq & \mathbf{let} \;\; z = [\![\, a \,]\!]^{\Phi} \; \mathbf{in} \; [\![\, a' \,]\!]^{\Phi,\, z \mapsto z} \qquad z \notin \mathrm{dom}(\Phi) \\
[\![\, \mathbf{letrec}_0 \;\; z = \lambda x.a \;\; \mathbf{in} \;\; a' \,]\!]^{\Phi} & \triangleq & [\![\, a' \,]\!]^{\Phi,\, z \mapsto [\![\, \mu(z,x,a) \,]\!]^{\Phi}} \qquad z \notin \mathrm{dom}(\Phi) \\
[\![\, \mathbf{letrec}_1 \;\; z = \lambda x.a \;\; \mathbf{in} \;\; a' \,]\!]^{\Phi} & \triangleq & \mathbf{letrec} \;\; z = [\![\, \lambda x.a \,]\!]^{\Phi,\, z \mapsto z} \; \mathbf{in} \; [\![\, a' \,]\!]^{\Phi,\, z \mapsto z} \\
 & & \qquad z \notin \mathrm{dom}(\Phi)
\end{array}
$$

We extend $[\![-]\!]^{-}$ to act on A_1 contexts by adding the clause $[\![\_]\!]^{\Phi} = \_$. On reduction contexts we define the action as:

$$
\begin{array}{rcl}
[\![\, \mathbf{let}_0 \;\; z = u \;\; \mathbf{in} \;\; \_.E_3 \,]\!]^{\Phi} & \triangleq & [\![\, E_3 \,]\!]^{\Phi,\, z \mapsto [\![\, u \,]\!]^{\Phi}} \\
[\![\, \mathbf{letrec}_0 \;\; z = \lambda x.a \;\; \mathbf{in} \;\; \_.E_3 \,]\!]^{\Phi} & \triangleq & [\![\, E_3 \,]\!]^{\Phi,\, z \mapsto [\![\, \mu(z,x,a) \,]\!]^{\Phi}} \\
[\![\, A_1.E_3 \,]\!]^{\Phi} & \triangleq & [\![\, A_1 \,]\!]^{\Phi}.[\![\, E_3 \,]\!]^{\Phi} \\
[\![\, \_.E_3 \,]\!]^{\Phi} & \triangleq & \_.[\![\, E_3 \,]\!]^{\Phi}
\end{array}
$$

Fig. A 3. Instantiate-substitute correspondence and its extension to evaluation contexts.

occur free in the environment have definitions further up the environment. Clause (iii) ensures that each identifier is defined only once, allowing us to treat an environment as a finite partial function without ambiguity.

The function $[\![-]\!]^{\Phi}$ is not well defined for all terms. Given a well-formed environment Φ the function $[\![-]\!]^{\Phi}$ on λ terms acts on variables by looking them up in the environment Φ. Thus, it is well defined only for terms whose free variables are contained in the domain of Φ. In addition, because recursive functions, $\lambda^z x.a$, mention a variable z, whenever we apply $[\![-]\!]^{\Phi}$ to such a term z should be mapped by Φ. In this case, the environment and the term both associate a function with z and we must ensure that the terms they associate with it are compatible. The correct definition of compatible is that the body of the function in the environment is the

image of the one in the term under $[\![-]\!]^{\Phi'}$, where Φ' is the bindings above z in Φ extended to map the free variables x and z to themselves. The following definition formalises this compatibility of environment and term.

Definition 5 (*Environment-term compatibility*)
A term a is compatible with an environment Φ, written $\Phi \blacktriangleleft a$, if and only if the following hold:

i. the environment is well formed: $\Phi \blacktriangleleft$
ii. $\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi)$
iii. for all $\lambda^z x.\hat{a} \in \mathrm{frf}(a)$ there exists Φ_1, Φ_2, e such that $\Phi = \Phi_1, z \mapsto \lambda x.\mathbf{letrec}\ z = \lambda x.e\ \mathbf{in}\ e, \Phi_2$ and $\Phi_1, x \mapsto x, z \mapsto z \blacktriangleleft \hat{a}$ and $e = [\![\hat{a}]\!]^{\Phi_1, x \mapsto x, z \mapsto z}$.

The definition extends naturally to evaluation contexts $\Phi \blacktriangleleft E_3$.

The following lemma shows that environment-term compatibility is closed under reduction.

Lemma 13 ($- \blacktriangleleft -$ *is closed under reduction*)
$\Phi \blacktriangleleft a \ \wedge\ a \longrightarrow_{r'} a' \implies \Phi \blacktriangleleft a'$ □

A.1.1 Defining R

We have now defined a function to relate the intermediate language $\lambda_{r'}$ to λ_c. However, we wish to define a candidate bisimulation relation, R, between λ_r and λ_c. We therefore need a way of relating unannotated λ_r and annotated $\lambda_{r'}$ terms. Fortunately, this is straightforward. Going from unannotated to annotated, it is assumed that the whole term is part of the program, so all **let** s are 1-annotated and functions are left unannotated, while the reverse direction is the forgetful function that removes all annotations. The former is called *inject* (and written $\iota[\_]$) and the latter *erase* (written $\epsilon[\_]$). Their rather simple definitions are omitted.

We can now give our definition of the candidate weak bisimulation R.

Definition 6 (*Candidate eventually weak bisimulation*)

$$R \equiv \{(e, e') \mid \exists\, a.\ \mathrm{wf}[a] \ \wedge\ a\ \ \mathsf{closed} \ \wedge\ e = [\![a]\!]^{\varnothing} \ \wedge\ e' = \epsilon[a]\}$$

This relation is defined on unannotated terms, but defined in terms of projections out of an annotated $\lambda_{r'}$ term; $[\![-]\!]^-$ forms the corresponding λ_c term and $\epsilon[-]$ the corresponding λ_r term.

The goal is to show that if we start with identical terms, then the two reduction systems reduce them to equivalent values, we therefore need identical terms to be related by R. We check this sanity property.

Definition 7 (id_λ)
The identity relation on closed λ terms is id_λ, that is: $\mathrm{id}_\lambda = \{(e, e) \mid e \in \lambda \ \wedge\ e\ \ \mathsf{closed}\}$, where λ is the set of all λ terms.

Lemma 14 (R *contains identity*)
The candidate bisimulation R contains id_λ. □

A.1.2 Basic properties of constituents of R

This section establishes some basic properties of $[\![-]\!]^-$, $\epsilon[-]$ and wf$[-]$—the basic building blocks of R—as well as environment well-formedness conditions. We are mainly interested in how the operations distribute over our syntax, that they preserve values and that well formedness of terms is preserved by reduction.

The following definition provides the link between $\lambda_{r'}$ evaluation contexts and environments Φ.

Definition 8 (*Binding context*)
$\mathscr{E}_c[E_3]^\Phi$ builds an environment corresponding to the binding context of the $\lambda_{r'}$ reduction context E_3 using the environment Φ.

$$\begin{array}{lcl}
\mathscr{E}_c[_]^\Phi & = & \emptyset \\
\mathscr{E}_c[_.E_3]^\Phi & = & \mathscr{E}_c[E_3]^\Phi \\
\mathscr{E}_c[A_1.E_3]^\Phi & = & \mathscr{E}_c[E_3]^\Phi \\
\mathscr{E}_c[\mathbf{let}_0\ z = u\ \mathbf{in}\ _.E_3]^\Phi & = & z \mapsto [\![\, u \,]\!]^\Phi, \mathscr{E}_c[E_3]^{\Phi,\, z \mapsto [u]^\Phi} \\
\mathscr{E}_c[\mathbf{letrec}_0\ z = \lambda x.a\ \mathbf{in}\ _.E_3]^\Phi & = & z \mapsto [\![\, \mu(z,x,a) \,]\!]^\Phi, \mathscr{E}_c[E_3]^{\Phi,\, z \mapsto [\mu(z,x,a)]^\Phi}
\end{array}$$

The context E_3 and the environment Φ must be compatible in the sense that $\mathrm{fv}(E_3) \subseteq \mathrm{dom}(\Phi)$ and $\mathrm{hb}(E_3)$ must be unique.

When extending an environment with a value care must be taken to ensure the resulting environment is well formed. The following facts are useful in doing this.

Proposition 1 (*Environment properties*)

i. If $\Phi \blacktriangleleft u$ and $z \notin \mathrm{dom}(\Phi)$, then $\Phi, z \mapsto [\![\, u \,]\!]^\Phi \blacktriangleleft$
ii. If $\Phi \blacktriangleleft a$ and $\Phi, \Phi' \blacktriangleleft$, then $\Phi, \Phi' \blacktriangleleft a$
iii. If $\Phi \blacktriangleleft E_3.a$, then $\Phi, \mathscr{E}_c[E_3]^\Phi \blacktriangleleft a$
iv. If $\Phi \blacktriangleleft a$ and wf$[a]$ and $a \longrightarrow_{r'} a'$, then $\Phi \blacktriangleleft a'$ □

We can extend parts (ii) and (iii) of the previous lemma to contexts to conclude the following.

Corollary 15 (*Environment context properties*)

i. If $\Phi \blacktriangleleft E_3$ and $\Phi, \Phi' \blacktriangleleft$, then $\Phi, \Phi' \blacktriangleleft E_3$
ii. If $\Phi \blacktriangleleft E_3.E_3'$, then $\Phi, \mathscr{E}_c[E_3]^\Phi \blacktriangleleft E_3'$ □

Lemma 16 ($[\![-]\!]^-$ *Value preservation*)
$\Phi \blacktriangleleft u \ \wedge\ \mathrm{wf}[u] \implies [\![\, u \,]\!]^\Phi\ \mathsf{cval}$ □

Lemma 17 (*Well-formed context decomposition*)
$\mathrm{wf}[E_3.a] \iff \mathrm{wf}[E_3] \ \wedge\ \mathrm{wf}[a]$ □

Lemma 18 ($\lambda_{r'}$ *reduction preserves well formedness*)
$\mathrm{wf}[a] \ \wedge\ a \longrightarrow_{r'} a' \implies \mathrm{wf}[a']$ □

We now prove some conditions under which a change of environment in $[\![-]\!]^-$ leaves the image unchanged.

Proposition 2 ($[\![-]\!]^-$ *Environment properties*)

i. If $\mathrm{wf}[a]$ and $\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi)$ and $\mathrm{fv}(v) \subseteq \mathrm{dom}(\Phi)$, then $\{v/x\}[\![a]\!]^{\Phi, x \mapsto x} = [\![a]\!]^{\Phi, x \mapsto v}$. If $\mathrm{wf}[a]$ and $\Phi, x \mapsto x \blacktriangleleft a$ and $\Phi, x \mapsto v, \Phi' \blacktriangleleft$, then $\{v/x\}[\![a]\!]^{\Phi, x \mapsto x, \Phi'} = [\![a]\!]^{\Phi, x \mapsto v, \Phi'}$

ii. If $x \notin \mathrm{fv}(a)$, then $[\![a]\!]^{\Phi, x \mapsto v} = [\![a]\!]^{\Phi}$. If $\Phi \blacktriangleleft a$ and $\Phi, \Phi' \blacktriangleleft a$, then $[\![a]\!]^{\Phi} = [\![a]\!]^{\Phi, \Phi'}$

iii. If $\Phi_1, \Phi_2, \Phi_3, \Phi_4 \blacktriangleleft a$ and $\Phi_1, \Phi_3, \Phi_2, \Phi_4 \blacktriangleleft a$, then $[\![a]\!]^{\Phi_1, \Phi_2, \Phi_3, \Phi_4} = [\![a]\!]^{\Phi_1, \Phi_3, \Phi_2, \Phi_4}$. □

Lemma 19 ($[\![-]\!]$ *Outer value preservation*)
For all $\lambda_{r'}$ values u:

a. If $\mathrm{wf}[u]$, $\Phi \blacktriangleleft u$ and $[\![u]\!]^{\Phi} = \lambda x{:}T.e$, then there exists E_2, a, j such that $u = E_2.\lambda^j x{:}T.a$
b. $[\![u]\!]^{\Phi} = (v_1, v_2) \implies \exists\, E_2, u_1, u_2.\ u = E_2.(u_1, u_2)$ □

Lemma 20 ($[\![-]\!]^-$ *Distribution over contexts*)
For all E_3, Φ and a, if $\Phi \blacktriangleleft E_3.a$ and $\mathrm{wf}[E_3.a]$, then $[\![E_3.a]\!]^{\Phi} = [\![E_3]\!]^{\Phi}.[\![a]\!]^{\Phi, \mathscr{E}_c[E_3]^{\Phi}}$
□

Lemma 21 ($[\![-]\!]$ *Preserves contexts*)
If $\Phi \blacktriangleleft E_3$ and $\mathrm{wf}[E_3]$, then there exists a λ_c reduction context E such that $[\![E_3]\!]^{\Phi} = E$.
□

We now establish a similar set of properties for $\epsilon[-]$, although the definition is considerably simpler making the proofs routine.

Lemma 22 ($\epsilon[-]$ *Value preservation*)
$\mathrm{wf}[u] \implies \epsilon[u]\ \mathsf{rval}$ □

Lemma 23 ($\epsilon[-]$ *Distributes over contexts*)
$\epsilon[E_3.a] = \epsilon[E_3].\epsilon[a]$ □

Lemma 24 ($\epsilon[-]$ *Preserves contexts*)
If $\mathrm{wf}[E_3]$, then there exists a λ_r reduction context E_3' such that $\epsilon[E_3] = E_3'$. □

Lemma 25 ($\epsilon[-]$ *Outer value preservation*)
For all $\lambda_{r'}$ values u:

a. If $\mathrm{wf}[u]$ and $\epsilon[u] = E_2.\lambda x{:}T.e$, then there exists $\hat{E}_2, a, z, j$ such that $u = \hat{E}_2.\lambda^j x{:}T.a$
b. $\epsilon[u] = E_2.(v_1, v_2) \implies \exists\, \hat{E}_2, u_1, u_2.\ u = \hat{E}_2.(u_1, u_2)$ □

A.2 R is an eventually weak bisimulation

In this section, we show that R, as defined in Definition 6, is an eventually weak bisimulation between λ_c and λ_r. To do this, we factor the problem into two EWS, one from λ_c to λ_r and the other in the reverse direction. These EWS are further factored through the annotated calculus $\lambda_{r'}$.

A.2.1 An eventually weak CR-simulation

Before we prove that R is an EWS from λ_c to λ_r, we observe some key facts about reduction in $\lambda_{r'}$ The first is that performing instantiation reductions to a term a leaves the image of a under $[\![-]\!]^\Phi$ unchanged.

Lemma 26 ($[\![-]\!]^-$ *Invariant under insts*)

$\mathrm{wf}[a] \wedge \Phi \blacktriangleleft a \wedge a \xrightarrow{\text{insts}}{}^{*}_{r'} a' \implies [\![a]\!]^\Phi = [\![a']\!]^\Phi$ □

Another important observation is that every contiguous sequence of instantiations is finite. That is, we eventually reach a term that cannot reduce via an instantiation. We say such a term is in instantiation normal form, (INF) which is formally defined as follows (the obvious variant of these definitions hold for λ_r as well).

Definition 9 (*INF*)

A term a is in *instantiation normal form* if and only if there does not exist an a' such that $a \xrightarrow{\text{insts}} a'$. We write $a \ \mathsf{inf}_\mathsf{r}$ when a is in INF.

Definition 10 (*Open INF*)

A possibly open term a is in *open instantiation normal form* if and only if there does not exist an E_3 and z such that $a = E_3.z$. We write $a \ \mathsf{inf}_\mathsf{r}^\circ$ when a is in open INF.

INF and open INF agree on closed terms, but not necessarily on open ones. For example, if there does not exist E_3, E_3', z, x, u, a such that $a = E_3.\mathbf{let}_0 \ z = u \ \mathbf{in} \ E_3'.z$ and $a \neq E_3'.\mathbf{letrec}_0 \ z = \lambda x.a \ \mathbf{in} \ E_3'.z$, then a cannot perform an inst or instrec reduction and is in INF. However, it may still be the case that for some E_3'' and z that $a = E_3''.z$ as long as $z \notin \mathrm{hb}(E_3'')$, and therefore a is not in open INF.

A useful property of instantiation normal forms is that they are preserved by removing a surrounding E_3 context, the proof of which follows easily by proving the contrapositive.

Lemma 27 (inf_r° *preserved by* E_3 *removal*)

For any evaluation context E_3, if $E_3.a \ \mathsf{inf}_\mathsf{r}^\circ$, then $a \ \mathsf{inf}_\mathsf{r}^\circ$ □

To prove that we can reach an instantiation normal form from any $\lambda_{r'}$ term by reduction, we observe that the number of variables above λ's decreases with every instantiation. Therefore, we define the function instvar[e] in Definition 11 that counts the number of variables above λ's and prove that this is monotonically decreasing w.r.t. instantiation reductions to obtain an 'INF reachability' result.

Definition 11 (instvar[−])
The function instvar[a] denotes the number of potential instantiations that a can do.

$$\begin{aligned}
\text{instvar}[z] &= 1\\
\text{instvar}[n] &= 0\\
\text{instvar}[()] &= 0\\
\text{instvar}[\pi_r\ a] &= \text{instvar}[a]\\
\text{instvar}[(a\ a')] &= \text{instvar}[a] + \text{instvar}[a']\\
\text{instvar}[\lambda^j x.a] &= 0\\
\text{instvar}[a\ a'] &= \text{instvar}[a] + \text{instvar}[a']\\
\text{instvar}[\mathbf{let}_m\ z = a\ \mathbf{in}\ a'] &= \text{instvar}[a] + \text{instvar}[a']\\
\text{instvar}[\mathbf{letrec}_m\ z = \lambda x.a\ \mathbf{in}\ a'] &= \text{instvar}[a']
\end{aligned}$$

Lemma 28 (instvar[−] *properties*)
For all $\lambda_{r'}$ terms a and a'

1. $a\ \mathsf{r'val} \implies \text{instvar}[a] = 0$
2. $a \xrightarrow{\text{insts}}_{r'} a' \implies \text{instvar}[a'] = \text{instvar}[a] - 1$ □

Lemma 29 (*INF reachability*)
For all closed a, if wf[a], then there exists a' such that $a \xrightarrow{\text{insts}^*}_{r'} a' \wedge a'\ \mathsf{inf_r}$ □

The next fact also highlights the importance of INF. We might imagine that if $[\![a]\!]^\Phi$ is a value, then a is a value, but a counter-example is quite easy to find:

$$[\![\ \mathbf{let}\ x = 3\ \mathbf{in}\ x\ ]\!]^\Phi = 3$$

The result 3 is a value, but **let** $x = 3$ **in** x is not. The extra requirement needed is that a is in INF.

Lemma 30 ($[\![-]\!]^\Phi$ *source-value property*)
For all $\lambda_{r'}$ expressions a, the following holds:

$$\text{wf}[a] \wedge a\ \mathsf{inf}_\mathsf{r}^\circ \wedge \Phi \blacktriangleleft a \wedge [\![a]\!]^\Phi\ \mathsf{cval} \implies a\ \mathsf{r'val}$$

□

We can now prove a vital correspondence between λ_c and $\lambda_{r'}$.

Lemma 31 ($c - r'$ *correspondence*)
If a closed and wf[a] and $[\![a]\!]^\varnothing \longrightarrow_c e'$, then there exists a', a'' such that $a \xrightarrow{\text{insts}^*}_{r'} a'' \longrightarrow_{r'} a'$ and $a''\ \mathsf{inf_r}$ and either

i. $e' = [\![a']\!]^\varnothing$ or
ii. there exists e'' such that $e' \longrightarrow_c e''$ and $e'' = [\![a']\!]^\varnothing$.

We now wish to prove a similar correspondence between $\lambda_{r'}$ and λ_r. To do so, we need two important lemmas.

Lemma 32 (*Inst match property*)

$$\mathrm{wf}[a] \ \wedge \ a \xrightarrow{\mathrm{insts}}_{r'} a' \implies \exists\, e'.\, \epsilon[a] \xrightarrow{\mathrm{insts}}_{r} e' \ \wedge \ e' = \epsilon[a']$$

□

Lemma 33 (*Inst match sequence*)

$$\mathrm{wf}[a] \ \wedge \ a \xrightarrow{\mathrm{insts}^{\,n}}_{r'} a' \implies \exists\, e'.\, \epsilon[a] \xrightarrow{\mathrm{insts}^{\,n}}_{r} e' \ \wedge \ e' = \epsilon[a']$$

□

Lemma 34 ($r' - r$ *correspondence*)

$$a \ \mathsf{closed} \ \wedge \ \mathrm{wf}[a] \ \wedge \ a \xrightarrow{1}_{r'} a' \ \wedge \ l \neq \mathrm{zero} \implies \exists\, e'.\, \epsilon[a] \longrightarrow_{r} e' \ \wedge \ e' = \epsilon[a']$$

Putting the $c - r'$ and $r' - r$ correspondences together and using the following lemma (easily proved by inspection), we obtain the cr-simulation result. □

Lemma 35 ($\epsilon[-]$ *invariant under zeros*)
$\mathrm{wf}[a] \ \wedge \ a \xrightarrow{\mathrm{zeros}^{\,*}}_{r'} a' \implies \epsilon[a] = \epsilon[a']$ □

Finally, we can prove that R is an EWS from λ_c to λ_r.

Lemma 36 (*cr eventually weak simulation*)
R is an eventually weak simulation from λ_c to λ_r □

A.2.2 An eventually weak RC-simulation

We now prove the reverse simulation using a similar process to the one used to prove the CR-simulation. The role played by INFs is replaced by zero normal forms (ZNFs), with zeros in $\lambda_{r'}$ matching **l**et-reductions in λ_c.

We first define ZNF, establish some properties of it and prove that these forms are always reachable. We do not need to define open and closed ZNFs as we did with INF as the two definitions coincide. That is, a term a cannot do an instantiation reduction if and only if the following ZNF condition holds:

Definition 12 (*Open ZNF*)
We say that a possibly open $\lambda_{r'}$ expression is in *open zero normal form* and write $a \ \mathsf{znf}_{\mathsf{r}}^{\circ}$ if and only if there does not exist E_3, z, u, a' such that $a = E_3.\mathbf{let}_1 \ z = u \ \mathbf{in} \ a'$

Lemma 37 ($\mathsf{znf}_{\mathsf{r}}^{\circ}$ *preserved by* E_3 *stripping*)
$E_3.a \ \mathsf{znf}_{\mathsf{r}}^{\circ} \implies a \ \mathsf{znf}_{\mathsf{r}}^{\circ}$ □

Lemma 38 ($\epsilon[-]$ *Source-value property*)
$\mathrm{wf}[a] \ \wedge \ a \ \mathsf{znf}_{\mathsf{r}}^{\circ} \ \wedge \ \epsilon[a] \ \mathsf{rval} \implies a \ \mathsf{r'val}$ □

Lemma 39 ($\epsilon[-]$ *source context*)
If $\epsilon[a] = E_3.e$ and $a \ \mathsf{znf}_{\mathsf{r}}^{\circ}$, then there exists an $\hat{E}_3$ and $\hat{a}$ such that $a = \hat{E}_3.\hat{a}$ and $\epsilon[\hat{E}_3] = E_3$. □

Lemma 40 (*ZNF reachability*)
For all closed a, if $\mathrm{wf}[a]$, then there exists a' such that $a \xrightarrow{\text{zero}}{}^{*}_{r'} a' \wedge a'\ \mathsf{znf_r}$ □

To see the above lemma, observe that all contiguous sequences of (zero)-reductions are finite. Define a metric ones: $\lambda' \to \mathbb{N}$ that counts the number of 1-annotated-**let** s in an expression, then each (zero) reduction strictly reduces this measure. As expressions are finite, our metric is finite valued and thus reduction sequences consisting only of (zero)-reductions are finite.

For every zero or zerorec reduction that $\lambda_{r'}$ can do, λ_c can match it. As noted by the following two lemmas, which can be proved by induction on the transition systems and number of reductions, respectively.

Lemma 41 (*Zero match property*)

$$\mathrm{wf}[a] \wedge \Phi \blacktriangleleft a \wedge a \xrightarrow{\text{zero}}_{r'} a' \implies \exists\, e'.\ [\![a]\!]^{\Phi} \xrightarrow{\text{let}}_{c} e' \wedge e' = [\![a']\!]^{\Phi}$$

□

Lemma 42 (*Zero match sequence*)

$$\mathrm{wf}[a] \wedge \Phi \blacktriangleleft a \wedge a \xrightarrow{\text{zero}}{}^{n}_{r'} a' \implies \exists\, e'.\ [\![a]\!]^{\Phi} \xrightarrow{\text{let}}{}^{n}_{c} e' \wedge e' = [\![a']\!]^{\Phi}$$

□

Lemma 43 ($r - r'$ *correspondence*)

$$\begin{aligned} a\ \mathsf{closed} \wedge \mathrm{wf}[a] \wedge \epsilon[a] \quad &\longrightarrow_r \quad e' \implies \exists\, a', a''.\ a \xrightarrow{\text{zero}}{}^{*}_{r'} a'' \\ &\longrightarrow_{r'} \quad a' \wedge a''\ \mathsf{znf_r} \wedge e' = \epsilon[a'] \end{aligned}$$

□

Lemma 44 ($r' - c$ *correspondence*)
If a closed and $\mathrm{wf}[a]$ and $a \xrightarrow{1}_{r'} a'$ and $l \neq \mathrm{insts}$, then there exists an e' such that $[\![a]\!]^{\varnothing} \longrightarrow_c e'$ and either:

(i) $e' = [\![a']\!]^{\varnothing}$ or
(ii) there exists e'' such that $e' \longrightarrow_c e''$ and $e'' = [\![a']\!]^{\varnothing}$ □

Lemma 45 (*r-c eventually weak simulation*)
R is an eventually weak simulation from λ_r to λ_c. □

A.3 Equivalence

Having demonstrated an eventually weak bisimulation between λ_c and λ_r, we now use that relation to establish observational equivalence. The EWB tells us how terms reduced under λ_r and λ_c are related. However, because the bisimulation is weak, it does not tell us anything about how termination behaviour is related to the two calculi.

We must show that the termination of expressions coincides for both systems in order to show that the two are observationally equivalent. We first relate $[\![-]\!]$ and $[\![-]\!]^-$. The former is used to obtain λ_c values from λ_r results (or, equivalently, erased $\lambda_{r'}$ results), while the latter provides the link between λ_c and $\lambda_{r'}$ expressions; we show they are consistent. The main difference is that $[\![-]\!]$ uses substitution, whereas $[\![-]\!]^-$ uses an environment. In what follows, we write σ to range over substitutions. The following definition introduces a function S that builds a substitution from an environment.

Definition 13 (*Environment-substitution correspondence*)

$$\begin{array}{lcl} \mathrm{S}[\Phi\,,\, z \mapsto [\![\, u \,]\!]^{\Phi}] & = & \mathrm{S}[\Phi]\{[\![\, \epsilon[u] \,]\!]/z\,\} \\ \mathrm{S}[\varnothing] & = & \{\} \end{array}$$

The simple value-collapsing function $[\![\, \epsilon[-] \,]\!]$ and $[\![-]\!]^-$ do not agree on all values, but only those of ground type. For values of function type, they may not agree as they may differ in their recursive unrollings:

$$[\![\, \epsilon[\mathbf{letrec}_0\ z =_\lambda x.a\ \ \mathbf{in}\ \lambda^z x.a] \,]\!] \quad = \quad \{\lambda x.\mathbf{letrec}\ z = \lambda x.\epsilon[a]\ \mathbf{in}\ \epsilon[a]/z\,\}(\lambda x.\epsilon[a])$$

but

$$[\![\, \mathbf{letrec}_0\ z =_\lambda x.a\ \ \mathbf{in}\ \lambda^z x.a \,]\!]^{\varnothing} \quad = \quad \lambda x.\mathbf{letrec}\ z = [\![\, a \,]\!]^{x \mapsto x\,,\, z \mapsto z}\ \mathbf{in}\ [\![\, a \,]\!]^{x \mapsto x\,,\, z \mapsto z}$$

It turns out that the results of these operators do agree above λ abstractions, which is sufficient for our purposes as we need to consider only contextual equivalence at integer type. This motivates the next definition that is used in following lemma to prove a compatibility result between the two functions.

Definition 14 (*Equality on λ terms up to functions*)
We define $=_\lambda$ to be the standard equality relation up to α-equivalence, but extended to equate every function.

Lemma 46 (*Value correspondence*)
If $\Phi = \Phi_k$ where

$$\begin{array}{lcl} \Phi_0 & = & \varnothing \\ \Phi_{n+1} & = & \Phi_n, x_{n+1} \mapsto [\![\, u_{n+1} \,]\!]^{\Phi_n} \quad \text{where } \mathrm{fv}([\![\, u_{n+1} \,]\!]^{\Phi_n}) = \varnothing \end{array}$$

and $\Phi \blacktriangleleft u$ and $\mathrm{wf}[u]$, then $\mathrm{S}[\Phi][\![\, \epsilon[u] \,]\!] =_\lambda [\![\, u \,]\!]^{\Phi}$. □

The following two facts about typing are easily proved by induction on the typing derivation.

Lemma 47 (*Typing is substitutive*)
$\Gamma \vdash v : T \ \wedge\ \Gamma, z : T \vdash e : T' \implies \Gamma \vdash \{v/z\}e : T'$ □

Lemma 48 ($[\![-]\!]$ *Type preservation*)
$\Gamma \vdash u : T \implies \Gamma \vdash [\![\, u \,]\!] : T$ □

A.3.1 Proof of the main theorem

Proof of Theorem 12

We are now in a position to prove the main theorem of this section. To prove the first part, we first prove the following fact:

$$e \ \mathsf{closed} \wedge e \longrightarrow_c^* v_1 \implies \exists\, v_2, u.\ e \longrightarrow_r^* v_2 \wedge \mathrm{wf}[u] \wedge u \ \mathsf{closed} \wedge v_1 = [\![u]\!]^{\emptyset} \wedge v_2 = \epsilon[u] (*)$$

Assume e closed and $e \longrightarrow_c^* v_1$, and recall $e \ R \ e$ by R contains identity (Lemma 14). By c-r eventually weak simulation (Lemma 36) R is a c-r simulation, thus there exists an e' such that $e \longrightarrow_r^* e'$ and $v_1 \ R \ e'$. Expanding the definition of R in the latter, we are ensured that

$$\exists\, a.\ \mathrm{wf}[a] \wedge a \ \mathsf{closed} \wedge v_1 = [\![a]\!]^{\emptyset} \wedge e' = \epsilon[a]$$

We are left to show $e' \longrightarrow_r^* e''$ and e'' rval. By $\epsilon[-]$ Source-value property (Lemma 38), it suffices to prove that there exists an a' such that a' r'val $\wedge$ wf$[a']$ $\wedge$ a' $\mathsf{znf}_{\mathsf{r}}$ $\wedge$ $e'' = \epsilon[a']$.

Suppose that a $\mathsf{inf}_{\mathsf{r}}$, then by $[\![-]\!]^{\Phi}$ Source-value property (Lemma 30), a r'val. By $\epsilon[-]$ value preservation (Lemma 22), $\epsilon[a]$ rval as required.

Now suppose that $\neg(a \ \mathsf{inf}_{\mathsf{r}})$, then by INF reachability (Lemma 29), there exists an a'' such that $a \longrightarrow_{r'}^* a' \wedge a' \ \mathsf{inf}_{\mathsf{r}}$. By $\lambda_{r'}$-reduction preserves well formedness (Lemma 18) wf$[a']$ and by $[\![-]\!]^{-}$ Invariant under Insts (Lemma 26), $v_1 = [\![a']\!]^{\emptyset}$. Thus, by $[\![-]\!]^{\Phi}$ Source-value property (Lemma 30), a' r'val. By Inst match sequence (Lemma 33), there exists an e'' such that $e' \longrightarrow_r^* e'' \wedge e'' = \epsilon[a']$ as required.

We now prove the following result:

$$\vdash e : \mathsf{int} \wedge e \longrightarrow_c^* n \implies \exists\, v.\ e \longrightarrow_r^* v \wedge n = [\![v]\!]$$

Assuming $\vdash e : T \wedge e \longrightarrow_c^* n$, we can derive e closed, thus by (*) we know that there exists a u and v_2 such that $e \longrightarrow_r^* v_2 \wedge \mathrm{wf}[u] \wedge u \ \mathsf{closed} \wedge n = [\![u]\!]^{\emptyset} \wedge v_2 = \epsilon[u]$.

We are left to show that $n = [\![v_2]\!]$. By Value correspondence (Lemma 46), $[\![\epsilon[u]]\!] = [\![u]\!]^{\emptyset}$. We are left to show that this value is an integer, for which it suffices to show that one of the values in the equality above types to int, as the only values of type int in λ_c are integers. By type preservation for $\lambda_r \vdash v_2 : \mathsf{int}$, thus $\vdash \epsilon[u] : \mathsf{int}$ by dint of equality with v_2. By $[\![-]\!]$ Type preservation (Lemma 48), $\vdash [\![\epsilon[u]]\!] : \mathsf{int}$, as required.

Now we prove the second part of the main theorem. As before, we first prove

$$e \ \mathsf{closed} \wedge e \longrightarrow_r^* v_1 \implies \exists\, v_2, u.\ e \longrightarrow_c^* v_2 \wedge \mathrm{wf}[u] \wedge u \ \mathsf{closed} \wedge v_2 = [\![u]\!]^{\emptyset} \wedge v_1 = \epsilon[u]$$

Assume e closed and $e \longrightarrow_r^* v_1$, and recall $e \ R \ e$ by R contains identity (Lemma 14). By r-c eventually weak simulation (Lemma 45) R is a r-c simulation, thus there exists an e' such that $e \longrightarrow_c^* e'$ and $e' \ R \ v_1$. Expanding the definition of R in the latter, we are ensured that

$$\exists\, a.\ \mathrm{wf}[a] \wedge a \ \mathsf{closed} \wedge e' = [\![a]\!]^{\emptyset} \wedge v_1 = \epsilon[a]$$

We are left to show $e' \longrightarrow_c^* e''$ and e'' cval. By $[\![-]\!]^\Phi$ Source-value property (Lemma 30), it suffices to prove that there exists an a' such that a' r'val $\wedge$ wf$[a']$ $\wedge$ a' inf_r $\wedge$ $e'' = [\![a']\!]^\varnothing$.

Suppose that a znf_r, then by $\epsilon[-]$ Source-value property (Lemma 38), a r'val. By $[\![-]\!]^-$ Value preservation (Lemma 16), $[\![a']\!]^\varnothing$ cval as required.

Now suppose that $\neg(a$ $\mathsf{znf}_\mathsf{r})$, then by ZNF reachability (Lemma 40), there exists an a'' such that $a \xrightarrow{\text{zeros}}{}^*_{r'} a'$ $\wedge$ a' znf_r. By $\lambda_{r'}$-reduction preserves well formedness (Lemma 18) wf$[a']$ and by $\epsilon[-]$ Invariant under zeros (Lemma 35), $v_1 = \epsilon[a']$. Thus, by $\epsilon[-]$ Source-value property (Lemma 38), a' r'val. By Zero match sequence (Lemma 42), there exists an e'' such that $e' \longrightarrow_c^* e''$ $\wedge$ $e'' = \epsilon[a']$ as required.

Now we prove the following result:

$$\vdash e : \mathsf{int} \wedge e \longrightarrow_r^* v \implies \exists\, n.\, e \longrightarrow_c^* n \wedge n = [\![v]\!]$$

Assume $\vdash e$:int and $e \longrightarrow_r^* v$, then by the above lemma there exists a v_2 and a u such that $e \longrightarrow_c^* v_2$; wf$[u]$; u closed; $v_2 = [\![u]\!]^\varnothing$; $v = \epsilon[u]$ and u r'val.

We are left to show that $[\![u]\!] = n$. By Value correspondence (Lemma 46)$[\![\epsilon[u]]\!] = [\![u]\!]^\varnothing$. We are left to show that this value is an integer, for which it suffices to show that one of the values in the equality above types to int, as the only values of type int in λ_c are integers. By type preservation for $\lambda_r \vdash v$:int, thus $\vdash \epsilon[u]$:int by dint of equality with v. By $[\![-]\!]$ Type preservation (Lemma 48), $\vdash [\![\epsilon[u]]\!]$:int, as required. □

References

Abadi, M., Cardelli, L., Curien, P-L. & Lèvy, J-J. (1990) Explicit substitutions. In *Proc. 17th POPL, ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. pp. 31–46.

Ajmani, S. (2004) A review of software upgrade techniques for distributed systems. Available at: `http://pmg.csail.mit.edu/~ajmani/papers/review.pdf`. Accessed Sept 2007.

Ajmani, S., Liskov, B. & Shrira, L. (2006) Modular software upgrades for distributed systems. In *Proc. ECOOP, the 20th European Conference on Object-Oriented Programming (Nantes, France), LNCS 4067*. New York: Springer, pp. 452–476.

Altekar, G., Bagrak, I., Burstein, P. & Schultz, A. (2005 August) OPUS: Online patches and updates for security. In *Proceedings of 14th USENIX Security Symposium*. USENIX, Berkeley, CA, USA. pp. 287–302.

Ariola, Z. M. & Blom, S. (2002) Skew confluence and the lambda calculus with letrec. *Ann. Pure Appl. Logic*, **117**(1–3), 97–170.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. & Wadler, P. (1995 January). A call-by-need lambda calculus. In *Proc. 22nd POPL: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Francisco)*. pp. 233–246.

Armstrong, J., Virding, R., Wikstrom, C. & Williams, M. (1996) *Concurrent Programming in Erlang*, 2nd ed. Englewood Cliffs, NJ, USA. Prentice Hall.

Barklund, J. & Virding, R. (1999 February) Erlang 4.7.3 reference manual DRAFT (0.7). Available at: `http://www.erlang.org/download/erl_spec47.ps.gz`. Accessed Sept 2007.

Baumann, A., Appavoo, J., Silva, D. Da, Krieger, O. & Wisniewski, R. (2004 October). Improving operating system availability with dynamic update. In *Proceedings of the*

Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS) (Boston). pp. 21–27.

Baumann, A., Appavoo, J., Silva, D. Da, Kerr, J., Krieger, O. & Wisniewski, R. W. (2005) Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference (Anaheim, CA)*. USENIX. pp. 279–291.

Bierman, G., Hicks, M., Sewell, P., Stoyle, G. & Wansbrough, K. (2003a August) Dynamic rebinding for marshalling and update, with destruct-time lambda. In *Proceedings of ICFP 2003: the 8th ACM SIGPLAN International Conference on Functional Programming (Uppsala)*. pp. 99–110.

Bierman, G., Hicks, M., Sewell, P., Stoyle, G. & Wansbrough, K. (2003b June) *Dynamic Rebinding for Marshalling and Update, With Destruct-Time λ*. Tech. Rept. 568. University of Cambridge Computer Lab. Available at: `http://www.cl.cam.ac.uk/~pes20/`. Accessed Sept 2007.

Bierman, G., Hicks, M., Sewell, P. & Stoyle, G. (2003c April). Formalizing dynamic software updating. In *Proceedings of USE 2003: The Second International Workshop on Unanticipated Software Evolution (Warsaw)*.

Billings, J. (2005) *A Bytecode Compiler for Acute*. Computer Science Tripos Part II Dissertation, University of Cambridge.

Billings, J., Sewell, P., Shinwell, M. & Strniša, R. (2006 September) Type-safe distributed programming for OCaml. In *Proc. ML'06, 2006 ACM SIGPLAN Workshop on ML*. pp. 20–31.

Boa. (n.d.) Boa webserver. Available at: `http://www.boa.org`. Accessed Sept 2007.

Boyapati, C., Liskov, B., Shrira, L., Moh, C.-H. & Richman, S. (2003 October). Lazy modular upgrades in persistent object stores. In *Proc. OOPSLA: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (Anaheim, CA)*. pp. 403–417.

Cardelli, L. & Gordon, A. D. (1998) Mobile ambients. In *Proc. FoSSaCS: 1st International Conference on Foundations of Software Science and Computation Structure, as part of ETAPS (Lisbon), LNCS 1378*. New York: Springer, pp. 140–155.

Carlsson, Richard, Gustavsson, Björn, Johannson, Erik, Lindgren, Thomas, Nyström, Svel-Olof, Pettersson, Mikael, & Virding, Robert. 2004 (Nov.). *Core Erlang 1.0.3 language specification*. `http://www.it.uu.se/research/group/hipe/cerl/`. Accessed Sept 2007.

Chen, H., Chen, R., Zhang, F., Zang, B. & Yew, P.-C. (2006) Live updating operating systems using virtualization. In *Proceedings of VEE: the 2nd International Conference on Virtual Execution Environments (Ottawa)*. New York: ACM, pp. 35–44.

Chothia, T. & Stark, I. (2000) A distributed pi-calculus with local areas of communication. In *Proceedings of HLCL: The 4th International Workshop on High-Level Concurrent Languages (Montreal), published as Electr. Notes Theor. Comput. Sci.* **41**(2). pp. 1–16.

Dami, L. (1998) A lambda-calculus for dynamic binding. *Theor. Comput. Sci.* **192**(2), 201–231.

dlopen. (n.d.) *POSIX* `dlopen` specification. Available at: `http://www.opengroup.org/onlinepubs/007904975/functions/dlopen.html`. Accessed Sept 2007.

Drossopoulou, S. & Eisenbach, S. (2002 June) Manifestations of dynamic linking. In *Proceedings of the 1st Workshop on Unanticipated Software Evolution (USE 2002)*. Available at: `http://slurp.doc.ic.ac.uk/pubs/manifestations-use02.pdf`. Accessed Sept 2007.

Duggan, D. (2000) Sharing in typed module assembly language. In *Proceedings of TIC: The 3rd International Workshop on Types in Compilation (Montreal), Revised Selected Papers, LNCS 2071*. New York: Springer, pp. 85–116.

Duggan, D. (2001) Type-based hot swapping of running modules. In *Proc. 5th ICFP: The ACM SIGPLAN International Conference on Functional Programming (Firenze)*. pp. 62–73.

Fabry, R. S. (1976) How to design a system in which modules can be changed on the fly. In *Proceedings of the International Conference on Software Engineering (ICSE)*. pp. 470–476.

Felleisen, M. & Friedman, D. P. (1987) Control operators, the SECD-machine, and the lambda calculus. In *Formal Description of Programming Concepts III*, Wirsing, M. (ed). North-Holland: Elsevier, pp. 193–219.

Felleisen, M. & Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.* **103**(2), 235–271.

Felleisen, M., Wand, M., Friedman, D. P. & Duba, B. F. (1988 July) Abstract continuations: A mathematical semantics for handling full functional jumps. In *ACM Conference on LISP and Functional Programming (Snowbird, Utah)*. pp. 52–62.

Fournet, C., Gonthier, G., Lévy, J-J., Maranget, L. & Rémy, D. (1996) A calculus of mobile agents. In *Proceedings of CONCUR '96: The 7th International Conference on Concurrency Theory (Pisa), LNCS 1119*. New York: Springer, pp. 406–421.

Frieder, O. & Segal, M. E. (1991) On dynamically updating a computer program: From concept to prototype. *J. Syst. Software* **14**(2), 111–128.

Garrigue, J. (1995) Dynamic binding and lexical binding in a transformation calculus. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming*. Singapore: World Scientific, 14 pp.

Gilmore, S., Kirli, D. & Walton, C. (1997) *Dynamic ML Without Dynamic Types*. Tech. Rept. ECS-LFCS-97-378. Dept. of Computer Science, The University of Edinburgh.

Goldberg, A. & Robson, D. (1989) *Smalltalk 80—The Language and Its Implementation*. Reading MA: Addison-Wesley.

Gunter, C. A., Rémy, D. & Riecke, J. G. (1995 June) A generalisation of exceptions and control in ML-like languages. In *Proceedings of FPCA '95: The ACM SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture (La Jolla, CA)*. pp. 12–23.

Gupta, D. (1994 November) *On-line Software Version Change*, Ph.D. thesis. Kanpur, India: Department of Computer Science and Engineering, Indian Institute of Technology.

Hashimoto, M. & Ohori, A. (2001) A typed context calculus. *Theor. Comput. Sci.* **266**(1–2), 249–272.

Hashimoto, M. & Yonezawa, A. (2000) MobileML: A programming language for mobile computation. In *Proc. COORDINATION (Limassol, Cyprus), LNCS 1906*. New York: Springer, pp. 198–215.

Hicks, M. (2001 August). *Dynamic Software Updating*, Ph.D. thesis. Philadelphia: University of Pennsylvania.

Hicks, M. & Weirich, S. (2000) *A Calculus for Dynamic Loading*. Tech. Rept. MS-CIS-00-07. Philadelphia: University of Pennsylvania.

Hicks, M., Weirich, S. & Crary, K. (2000) Safe and flexible dynamic linking of native code. In *Proceedings of TIC: the 3rd International Workshop on Types in Compilation (Montreal), Revised Selected Papers, LNCS 2071*. New York: Springer, pp. 147–176.

Hirschowitz, T. (2003) *Modules mixins, modules et récursion étendue en appel par valeur*, Thèse de doctorat. Université Paris 7.

Hirschowitz, T. Leroy, X. & Wells, J. B. (2003 August) Compilation of extended recursion in call-by-value functional languages. In *Proceedings of PPDP: the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (Uppsala)*. pp. 160–171.

Jagannathan, S. (1994) Metalevel building blocks for modular systems. *ACM Trans. Program. Lang. Syst.* **16**(3), 456–492.

Java. (n.d.) *Java platform debugger architecture*. (This supports class replacement). Available at: `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`. Accessed Sept 2007.

Kiselyov, O. Chieh Shan, C. & Sabry, A. (2006) Delimited dynamic binding. In *Proceedings of ICFP: the 11th ACM SIGPLAN International Conference on Functional Programming (Portland, Oregon)*. pp. 26–37.

Lee, I. (1983 April) *DYMOS: A dynamic modification system*. Ph.D. thesis. Madison: Department of Computer Science, University of Wisconsin.

Lee, S-D. & Friedman, D. P. (1993 January) Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proceedings of POPL: The 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston)*. pp. 479–492.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D. & Vouillon, J. (2001 December) *The Objective Caml System Release 3.04 Documentation*. Paris: Institut National de Recherche en In formatique et en Automatique.

Lewis, J. R., Launchbury, J., Meijer, E. & Shields, M. (2000 January) Implicit parameters: Dynamic scoping with static types. In *Proceedings of POPL: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Boston)*. pp. 108–118.

MIT. (n.d.) MIT Scheme. Available at: `http://www.swiss.ai.mit.edu/projects/scheme/`. Accessed Sept 2007.

Moreau, L. (1998) A syntactic theory of dynamic binding. *Higher-order Symbolic Comput.* **11**(3), 233–279.

Moreau, L. & Queinnec, C. (1994) Partial continuations as the difference of continuations: A duumvirate of control operators. In *Proc. PLILP: The 6th International Symposium on Programming Language Implementation and Logic Programming (Madrid), LNCS 844*. New York: Springer, pp. 182–197.

Neamtiu, I. Hicks, M. Stoyle, G. & Oriol, M. (2006 June) Practical dynamic software updating for C. In *Proceedings of PLDI: The ACM Conference on Programming Language Design and Implementation (Ottawa)*. pp. 72–83.

Needham, R. M. (1993) Names. In *Distributed Systems*, Mullender, S. (ed) 2nd ed. Wokingham, England: Addison-Wesley, pp. 315–327.

Pai, V. S., Druschel, P. & Zwaenepoel, W. (1999 June) Flash: An efficient and portable webserver. In *Proceedings of the USENIX Annual Technical Conference*. pp. 106–119.

Peterson, J. Hudak, P. & Ling, G. S. (1997 July) *Principled Dynamic Code Improvement*. Tech. Rept. YALEU/DCS/RR-1135. New Haven, CT: Department of Computer Science, Yale University.

Potter, S. & Nieh, J. (2005 December) Reducing downtime due to system maintenance and upgrades. In *Proceedings of LISA: The 19th Conference on Systems Administration (San Diego)*. 47–62.

Queinnec, C. (1993) A library of high level control operators. *Lisp Pointers ACM SIGPLAN Spec. Interest Publ. Lisp* **6**(4), 11–26.

Riely, J. & Hennessy, M. (1999 January). Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Antonio)*. pp. 93–104.

Rouaix, F. (1996) A web navigator with applets in Caml. *Comput. Networks ISDN Sys.* **28**(7–11), 1365–1371.

Schmitt, A. (2002) Safe dynamic binding in the join calculus. In *Proceedings of IFIP TCS: IFIP International Conference on Theoretical Computer Science (Montréal). IFIP Conference Proceedings*, vol. 223. Norwell, MA: Kluwer, pp. 563–575.

Serra, A. Navarro, N. & Cortes, T. (2000) DITools: Application-level support for dynamic extension and flexible composition. In *Proc. USENIX Annual Technical Conference*. pp. 225–238.

Sewell, P. (1997) On implementations and semantics of a concurrent programming language. In *Proceedings of CONCUR 97: Concurrency Theory (Warsaw). LNCS 1243*. Berlin: Springer-Verlag, pp. 391–405.

Sewell, P. & Vitek, J. (2000) Secure composition of untrusted code: Wrappers and causality types. In *Proc. CSFW: The 13th IEEE Computer Security Foundations Workshop (Cambridge)*. pp. 269–284.

Sewell, P. Wojciechowski, P. T. & Pierce, B. C. (1999) Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages, LNCS 1686*. Springer, pp. 1–31.

Sewell, P. Leifer, J. J., Wansbrough, K. Allen-Williams, M. Zappa Nardelli, F. Habouzit, P. & Vafeiadis, V. (2004 October) *Acute: High-level Programming Language Design for Distributed Computation. Design Rationale and Language Definition*. Tech. Rept. UCAM-CL-TR-605. University of Cambridge Computer Laboratory. Also published as INRIA RR-5329. 193 pp.

Sewell, P. Leifer, J. J., Wansbrough, K. Zappa Nardelli, F. Allen-Williams, M. Habouzit, P. & Vafeiadis, V. (2007) Acute: High-level programming language design for distributed computation. *J. Funct. Programming* **17**(4–5), 547–612. Invited submission for an ICFP 2005 special issue.

Soules, C., Appavoo, J., Hui, K., Silva, D. Da, Ganger, G., Krieger, O., Stumm, M., Wisniewski, R., Auslander, M., Ostrowski, M., Rosenburg, B. & Xenidis, J. (2003 June) System support for online reconfiguration. In *Proceedings of the USENIX Annual Technical Conference (San Antonio)*. pp. 141–154.

Squeak. (n.d.) Squeak Smalltalk-80 Programming system. Available at: `http://www.squeak.org`

Stoyle, G. (2006) *A Theory of Dynamic Software Updates*, Ph.D. thesis. University of Cambridge.

Stoyle, G. Hicks, M. Bierman, G. Sewell, P. & Neamtiu, I. (2005 January) *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *Proc. POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Long Beach)*. pp. 183–194.

Vivas Frontana, J. L. (2001 March) *Dynamic Binding of Names in Calculi for Mobile Processes*, Ph.D. thesis. Stockholm: KTH.

Walker, D., Crary, K. & Morrisett, G. (2000) Typed memory management via static capabilities. *ACM Trans. Programming Lang. Syst.* **22**(4), 701–771.

Walton, C. (2001) *Abstract Machines for Dynamic Computation*, Ph.D. thesis. University of Edinburgh. ECS-LFCS-01-425.

Welsh, M. Culler, D. & Brewer, E. (2001 October) SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of SOSP: The 18th Eighteenth Symposium on Operating Systems Principles (Banff)*. pp. 230–243.