Technical Report

UCAM-CL-TR-568
ISSN 1476-2986

Number 568

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Dynamic rebinding for marshalling and update, with destruct-time λ

Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, Keith Wansbrough

February 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

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

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

†University of Cambridge
{First.Last}@cl.cam.ac.uk

‡University of Maryland, College Park
mwh@cs.umd.edu

Abstract

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

In this paper we adopt a foundational approach, developing core dynamic rebinding mechanisms as extensions to 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 thereby establish primitives and a common semantic foundation for a variety of real-world dynamic rebinding requirements.

Contents

List of Figures

LIST OF FIGURES

1 Introduction

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

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

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

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

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

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

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

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

In the remainder of the introduction we give an overview of our work, presented in §2–4. Relationships with prior work, and further discussion of the design space, are in §5; in §6 we comment on future work and conclude. Proofs of results are given in the Appendices. This technical report is an extended version of the paper [BHS+03], with differences as follows: in §2 the typing and runtime error rules are included, and additional examples given; in §3 the error rules for λ_{marsh} are included and the extension with distributed communication is fleshed out with examples, typing and semantics; and the appendices give proofs of the results for λ_c, λ_r, λ_d and λ_{marsh}.

Corrigendum Theorem 4 of the paper [BHS+03] asserted the observational equivalence of the three calculi λ_c, λ_r, and λ_d, as a check that the latter two are essentially call-by-value despite their rather different evaluation strategies. After publication, we discovered a technical flaw in the proof, which was based on an intricate operational correspondence argument. We conjecture that the original statement does hold, but have not proved it. Instead, in this Technical Report we state and prove the property for a simpler language, replacing **letrec** by a nonterminating Ω (with $\Omega \longrightarrow \Omega$).

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

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

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

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

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

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

Dynamic Rebinding: the λ_{marsh} Calculus With λ_d in place we can consider dynamic rebinding of marshalled values. The key question is this: when a value is moved between scopes, how can the user specify which identifiers should be rebound and which should be fixed? Our answer is embodied in the λ_{marsh} calculus of §3, which contains primitives for packaging a value such that some of its identifiers are fixed to bindings in the current context, while others will be rebound when unpackaged in a new scope (*e.g.*, when the value is moved). Which bindings will be fixed is dynamically determined with respect to a *mark*. Marking is done with an expression form

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

Here the mark name M is taken from a new syntactic class (not subject to binding); it names the surrounding declaration context. Packaging and unpackaging is done by expressions

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

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:

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

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

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

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

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

The **update** expression indicates that an update is possible at the point during evaluation when **update** appears in redex position. At that run-time point the user can supply an update of the form $\{w \Leftarrow e\}$, indicating that w should be rebound to expression e. In the example this update is $\{y \Leftarrow (x_1, 6)\}$; the let-binder for y_1 is modified accordingly yielding the expression on the right above, and thence a final result of 5. Here any identifier in scope at the update point can be rebound, to an expression that may mention identifiers in scope at its binding point. We define what it means for an update to be well-typed with respect to a program; applying well-typed updates preserves typing. The use of λ_d enables us to deal simply and cleanly with higher-order functions, largely ignored in past work. We imagine λ_{update} will form the core of future calculi that include other desirable features, such as state transformation, abstract types, changing the types of variables, multi-threading, etc. As a first step, in [BHSS03] we develop a model of updating in the style of Erlang [AVWW96].

2 Call-by-value λ-calculus revisited

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

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

$\boxed{\Gamma \vdash e{:}T}$

$$\frac{}{\Gamma, z{:}T, \Gamma' \vdash z{:}T}$$

$$\frac{}{\Gamma \vdash n{:}\mathsf{int}} \qquad \frac{\Gamma \vdash e{:}T \quad \Gamma \vdash e'{:}T'}{\Gamma \vdash (e, e'){:}T * T'} \qquad \frac{\Gamma \vdash e{:}T_1 * T_2}{\Gamma \vdash \pi_r\ e{:}T_1}$$

$$\Gamma \vdash (){:}\mathsf{unit}$$

$$\frac{\Gamma, z{:}T \vdash e{:}T'}{\Gamma \vdash \lambda z{:}T.e{:}T \to T'} \qquad \frac{\Gamma \vdash e'{:}T \to T' \quad \Gamma \vdash e{:}T}{\Gamma \vdash e'\,e{:}T'} \qquad \frac{\Gamma \vdash e{:}T \quad \Gamma, z{:}T \vdash e'{:}T'}{\Gamma \vdash \mathbf{let}\ z = e\ \mathbf{in}\ e'{:}T'}$$

$$\frac{\Gamma, z{:}T \to T', x{:}T \vdash e{:}T' \quad \Gamma, z{:}T \to T' \vdash e'{:}T''}{\Gamma \vdash \mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ e'{:}T''}$$

Figure 1: Lambda Calculi – Typing

where r ranges over $\{1, 2\}$. Expressions are taken up to alpha equivalence (though contexts are not). It is simply-typed, with a typing judgement $\Gamma \vdash e{:}T$ defined as usual, where Γ ranges over sequences of $z{:}T$ pairs containing at most one such for any z. The (standard) typing rules are given in Figure 1.

2.1 Construct-time

The standard semantics, here called the *construct-time* semantics, is recalled at the top of Figure 3. 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, *e.g.*, $E.E'$ and $E.e$, instead of the usual heavier brackets $E[e]$. Standard capture-avoiding substitution of e for z in e' is written $\{e/z\}e'$. We write $\mathrm{hb}(E)$, defined below, for the list of binders around the hole of E. For now we will be concerned only with the behaviour of closed expressions, without external library functions. The choice of a small-step semantics will be important when we add dynamic rebinding and communication later.

2.2 Redex-time

The redex-time and destruct-time semantics are also shown in Figure 3. Instead of substituting bindings of identifiers to values, as in the construct-time (app) and (let), both semantics introduce a **let** to record a binding of the abstraction's formal parameter to the application argument, *e.g.*,

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

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

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

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

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

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

Example (2) in Figure 2 illustrates identifier instantiation. While the construct-time strategy substitutes for x immediately, the redex-time strategy instantiates x under the **let**, following the evaluation order. Both this and the first example also illustrate a further aspect of the redex-time calculus: values u include let-bindings of the form $\textbf{let } z = u \textbf{ in } u'$. Intuitively, this is because a value should 'carry its bindings with it' preventing otherwise stuck applications, *e.g.*, $(\lambda x{:}\mathsf{int}.x)(\textbf{let } z = 3 \textbf{ in } 5)$ or (for an example where the **let** is not garbage) $(\lambda f{:}(\mathsf{int} \to \mathsf{int}).x\ 2)(\textbf{let } z = 3 \textbf{ in } \lambda x{:}\mathsf{int}.z)$. Note that identifiers are not values, so z, (z, z) and $\textbf{let } z = 3 \textbf{ in } (z, z)$ are not values. Values may contain free identifiers under lambdas, as usual, so $\lambda x{:}\mathsf{int}.z$ is an open value and $\textbf{let } z = 3 \textbf{ in } \lambda x{:}\mathsf{int}.z$ 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 lambda; terms such as $\pi_1\, (\textbf{let } z = 3 \textbf{ in } (4, 5))$ or (more interestingly) $\pi_1\, (\textbf{let } z = 3 \textbf{ in } (\lambda x{:}\mathsf{int}.z, 5))$ would otherwise be stuck. The side condition for (app) can always be achieved by alpha conversion; it prevents capture.

2.3 Destruct-time

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

To instantiate as late as possible, while remaining call-by-value, we instantiate only identifiers that are immediately under a projection or on the left-hand-side of an application. In these 'destruct' positions their values are about to be deconstructed, and so their outermost pair or lambda structure must be made manifest. The *destruct contexts* $R ::= \pi_r\ \_ \mid \_u$ can be seen as the outer parts of the construct-time (proj)

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

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

Figure 2: Call-by-Value Lambda Calculi Examples

and (app) redexes. The choice of destruct contexts is determined by the basic redexes – for example, if we added arithmetic operations, we would need to instantiate identifiers of int type before using them.

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

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

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

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

Further, we must be able to instantiate under nested bindings between the binding in question and its use. Therefore, (inst-2) must allow additional bindings E_2 and E_2' between R and the **let** and between the **let** and z. Similarly, (inst-1) must allow bindings E_2 between the R and z; it must allow both binding and evaluation contexts E_3 between the **let** and the R, *e.g.*, for the instance

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

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

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

2.4 Properties

This subsection gives properties of our various λ-calculi: sanity checks to confirm that our definitions are coherent and more substantial results showing that λ_r and λ_d are essentially CBV. Details of the proofs can be found in the appendices.

Construct-time λ_c

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

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

Redex-time λ_r

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

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

Destruct-time λ_d

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

$$\begin{array}{llll}
\text{(proj)} & \pi_r\,(E_2.(u_1, u_2)) & \longrightarrow & E_2.u_r \\
\text{(app)} & (E_2.(\lambda z{:}T.e))u & \longrightarrow & E_2.\textbf{let } z = u \textbf{ in } e \quad \text{if } \mathrm{fv}(u) \notin \mathrm{hb}(E_2) \\
\text{(inst-1)} & \textbf{let } z = u \textbf{ in } E_3.R.E_2.z & \longrightarrow & \textbf{let } z = u \textbf{ in } E_3.R.E_2.u \\
 & \quad \text{if } z \notin \mathrm{hb}(E_3, E_2) \text{ and } \mathrm{fv}(u) \notin z, \mathrm{hb}(E_3, E_2) & & \\
\text{(inst-2)} & R.E_2.\textbf{let } z = u \textbf{ in } E_2'.z & \longrightarrow & R.E_2.\textbf{let } z = u \textbf{ in } E_2'.u \\
 & \quad \text{if } z \notin \mathrm{hb}(E_2') \text{ and } \mathrm{fv}(u) \notin z, \mathrm{hb}(E_2') & & \\
\text{(instrec-1)} & \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_3.R.E_2.z & \longrightarrow & \textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_3.R.E_2.\lambda x{:}T.e \\
 & \quad \text{if } z \notin \mathrm{hb}(E_3, E_2) \text{ and } \mathrm{fv}(\lambda x{:}T.e) \notin \mathrm{hb}(E_3, E_2) & & \\
\text{(instrec-2)} & R.E_2.\textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_2'.z & \longrightarrow & R.E_2.\textbf{letrec } z = \lambda x{:}T.e \textbf{ in } E_2'.\lambda x{:}T.e \\
 & \quad \text{if } z \notin \mathrm{hb}(E_2') \text{ and } \mathrm{fv}(\lambda x{:}T.e) \notin \mathrm{hb}(E_2') & &
\end{array}$$

Figure 3: Three Call-by-Value Lambda Calculi

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 $\quad w \quad ::= \quad n \mid () \mid (u, u') \mid \lambda z{:}T.e$

(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 $\quad w \quad ::= \quad n \mid () \mid (u, u') \mid \lambda z{:}T.e \mid z$

(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 $\neg\exists z \in \mathrm{hb}(E_3, E_2).w = z$

Figure 4: Three Call-by-Value Lambda Calculi – Error Rules

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

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

Note that the destruct-time error rules defining e err, 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 type preservation and type safety properties for the three calculi.

Theorem 2 (Type preservation for λ_c, λ_r and λ_d) *If $\Gamma \vdash e{:}T$ and $e \longrightarrow e'$ then $\Gamma \vdash e'{:}T$.*

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

Finally we would like to show that all three calculi are observationally equivalent, and hence that both λ_r and λ_d are essentially call-by-value. As stated in the introduction, after publication of [BHS+03] we discovered a technical flaw in the proof of our original theorem, which was based on an intricate operational correspondence argument. We conjecture that the original statement does hold, but have not proved it. Instead, in this Technical Report we state and prove the property for a simpler language, replacing **letrec** by a nonterminating Ω (with $\Omega \longrightarrow \Omega$). The proof remains non-trivial; it involves constructing a tight correspondence between reduction steps in the three calculi. As we noted earlier, values in λ_r and λ_d may need to be 'cleaned-up' to exactly correspond to λ_c values.

Theorem 4 (Observational Equivalence) *For the calculi with Ω replacing **letrec**:*

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

Proof Sketch The proof technique is the same for both claims: generalise the claim to 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 simulation proofs is that of instantiation normal form. Essentially a term is in instantiation normal form if it can not do an instantiation reduction. It is important that this form is always finitely reachable by reduction from any term. Finally, we use the bisimulation and some auxiliary lemmas to prove the generalised claim.

3 A Dynamic Rebinding Calculus: λ_{marsh}

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

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

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

In this section we develop a calculi to support all of the above. The calculus λ_{marsh} extends the destruct-time λ_d-calculus of §2.3 with high-level representations of *marshalled* values and primitives to manipulate them. We make two main choices. First, to have as intuitive a semantics as possible we want dynamic rebinding to only occur when unmarshalling values, not during normal computation. Second, to allow the programmer to cleanly and flexibly notate which definitions should be fixed and which should be rebindable, we introduce *marks*

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

which name contexts. Marshal and unmarshal operations

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

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 cross that **mark** M (thus making them rebindable). When the packaged value is unpackaged by an **unmarshal** M' _, the latter identifiers are rebound to binders outside the closest enclosing **mark** M'.

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

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

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	$::=$	$z_i \mid n \mid () \mid (e, e') \mid \pi_r\ e \mid \lambda x_i{:}T.e \mid ee' \mid$
			$\textbf{let}\ z_k{:}T = e\ \textbf{in}\ e' \mid \textbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \textbf{in}\ e' \mid$
			$\textbf{mark}\ M\ \textbf{in}\ e \mid \textbf{marshal}\ M\ e \mid \textbf{marshalled}\ \Gamma\ u \mid \textbf{unmarshal}\ M\ e$

Example

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

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

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

$\xrightarrow{\text{(inst-1)}}$

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

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

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

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

Figure 5: Dynamic Rebinding Calculus λ_{marsh}: Syntax and Example

3.1 Syntax

The λ_{marsh} syntax and an example, discussed below, are given in Figure 5; the new semantic rules are given in Figures 6 and 8. The calculus requires a more elaborate treatment of alpha equivalence than λ_d. There – as usual for λ-calculi – we had to use alpha equivalence during normal computation steps, to avoid mistaken capture of identifiers as the rules move subterms between different scopes. Here that is still required, but occurrences of the 'same' identifier under different bindings must be related so that the identifier can be marshalled with respect to one and unmarshalled with respect to another. Accordingly, instead of working with identifiers x, we work with *variables* x_i that are pairs of an identifier x and a *tag* i, similar to the external and internal names used in some module systems. Alpha equivalence changes only the tags; tags for different identifiers lie in different namespaces, so *e.g.*,

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

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. Binding is as follows:

Term	Binding
$\lambda x_i{:}T.e$	x_i binds in e
let $z_k{:}T = e'$ **in** e	z_k binds in e
letrec $z_k{:}T' = \lambda x_i{:}T.e'$ **in** e	x_i binds in e' and the z_k binds in $\lambda x_i{:}T.e'$ and e
marshalled Γ u	Free z_k of u are bound by occurrences of z_k in the domain of Γ (for well-typed terms fv(**marshalled** Γ u) will always be empty).

Note that the argument u of **marshalled** Γ u is required to be a value.

As a minor variant, one could take a single namespace of tags, *e.g.* for $\lambda x_i{:}T.e$ having i binding in e. That would be technically slightly simpler, but examples would be cluttered by many different tags.

3.2 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 $\boxed{\textit{redexes}}$ 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))

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

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

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

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

3.3 Semantics

Turning now to the details of the rules, the (proj), (app) and (inst-r) rules are as in λ_d but with z_k instead of z. In the (marshal) and (unmarshal) rules we abuse notation, writing the context **mark** M **in** _ as **mark** M. The (marshal) rule copies all bindings and marks between the **marshal** M _ and the closest enclosing **mark** M, using the bindmark(_) auxiliary to extract the bind and mark components of a context E_3, discarding the evaluation context components: bindmark(_) = _, bindmark($E_3.A_1$) = bindmark(E_3), and bindmark($E_3.A_2$) = bindmark(E_3).A_2. The predicate dhb(E_3) holds iff the hole-binders of E_3 are all distinct (which can always be made so by alpha conversion). The auxiliary env(E_3) extracts the type environment of the hole-binders of E_3, so they can be recorded in the **marshalled** value. This and other auxiliary functions are collected in Figure 7.

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

The new rules are:

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

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

(proj)	$\pi_r\ (E_2.(u_1, u_2))$	$\rightharpoonup$	$E_2.u_r$
(app)	$(E_2.(\lambda z_k{:}T.e))u$	$\rightharpoonup$	$E_2.\textbf{let } z_k{:}T = u \textbf{ in } e$
	if $\mathrm{fv}(u) \notin \mathrm{hb}(E_2)$		
(inst-1)	$\textbf{let } z_k{:}T = u \textbf{ in } E_3.R.E_2.z_k$	$\rightharpoonup$	$\textbf{let } z_k{:}T = u \textbf{ in } E_3.R.E_2.u$
	if $z_k \notin \mathrm{hb}(E_3, E_2)$ and $\mathrm{fv}(u) \notin z_k, \mathrm{hb}(E_3, E_2)$		
(inst-2)	$R.E_2.\textbf{let } z_k{:}T = u \textbf{ in } E_2'.z_k$	$\rightharpoonup$	$R.E_2.\textbf{let } z_k{:}T = u \textbf{ in } E_2'.u$
	if $z_k \notin \mathrm{hb}(E_2')$ and $\mathrm{fv}(u) \notin z_k, \mathrm{hb}(E_2')$		

(instrec-1)
$\textbf{letrec } z_k{:}T' = \lambda x_i{:}T.e \textbf{ in } E_3.R.E_2.z_k \rightharpoonup \textbf{letrec } z_k{:}T' = \lambda x_i{:}T.e \textbf{ in } E_3.R.E_2.\lambda x_i{:}T.e$
if $z_k \notin \mathrm{hb}(E_3, E_2)$ and $\mathrm{fv}(\lambda x_i{:}T.e) \notin \mathrm{hb}(E_3, E_2)$
(instrec-2)
$R.E_2.\textbf{letrec } z_k{:}T' = \lambda x_i{:}T.e \textbf{ in } E_2'.z_k \rightharpoonup R.E_2.\textbf{letrec } z_k{:}T' = \lambda x_i{:}T.e \textbf{ in } E_2'.\lambda x_i{:}T.e$
if $z_k \notin \mathrm{hb}(E_2')$ and $\mathrm{fv}(\lambda x_i{:}T.e) \notin \mathrm{hb}(E_2')$

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

Figure 6: Dynamic Rebinding Calculus λ_{marsh}: Semantics

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

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

(writing snoc with a comma).
Define the list of typed hole-binders of E_3, written $\mathrm{thb}(E_3)$, by:

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

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 $\mathrm{dhb}(E_3, X) \implies \mathrm{dhb}(E_3.A_1, X)$.

Figure 7: Dynamic Rebinding Calculus λ_{marsh}: Auxiliary Functions

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

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

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

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

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

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

3.4 Typing and Run-Time Errors

In some cases one would expect dynamic rebinding to require a run-time check to ensure safety, *e.g.*, if code is sent to a site that may or may not provide some resource it requires. For λ_{marsh} we have new run-time errors, if a **marshal** or an **unmarshal** refers to a mark which is not in scope, or if at (unmarshal)-time the environment does not have the required binders at the correct types. At the very least, however, one would like a type system to exclude all run-time errors except these. This can be done by a simple type system (collected in Figure 9), as usual but with a type $\mathsf{Marsh}\ T$ of marshalled

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 $\neg\exists z_k \in \text{hb}(E_2, E_3).w = z_k$	
(app-err)	$E_3.(E_2.w)u$	err
	if $\neg\exists(\lambda x_i{:}T.e).w = \lambda x_i{:}T.e$ and $\neg\exists z_k \in \text{hb}(E_2, E_3).w = z_k$	
(grab-err)	$E_3.\textbf{marshal}\ M\ u$	err'
	if $\textbf{mark}\ M$ 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 $\neg\exists z_k \in \text{hb}(E_2, E_3).w = z_k$	
(ungrab-err2)	$E_3.\textbf{unmarshal}M.E_2.\textbf{marshalled}\ \Gamma\ u$	err'
	if $\textbf{mark}\ M$ 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 $\text{rebind}(\Gamma, \text{thb}(E_3))$ is not defined	

Figure 8: Dynamic Rebinding Calculus λ_{marsh}: Error Rules

$\boxed{\Gamma \vdash e{:}T}$

$$\frac{}{\Gamma, x_i{:}T \vdash x_i{:}T}$$

$$\frac{}{\Gamma \vdash n{:}\mathsf{int}} \qquad \frac{\Gamma \vdash e{:}T \quad \Gamma \vdash e'{:}T'}{\Gamma \vdash (e, e'){:}T * T'} \qquad \frac{\Gamma \vdash e{:}T_1 * T_2}{\Gamma \vdash \pi_1\ e{:}T_1}$$

$$\Gamma \vdash (){:}\mathsf{unit} \qquad \Gamma \vdash \pi_2\ e{:}T_2$$

$$\frac{\Gamma, x_i{:}T \vdash e{:}T'}{\Gamma \vdash \lambda x_i{:}T.e{:}T \to T'} \qquad \frac{\Gamma \vdash e'{:}T \to T' \quad \Gamma \vdash e{:}T}{\Gamma \vdash e'\,e{:}T'} \qquad \frac{\Gamma \vdash e{:}T \quad \Gamma, x_i{:}T \vdash e'{:}T'}{\Gamma \vdash \textbf{let}\ x_i{:}T = e\ \textbf{in}\ e'{:}T'}$$

$$\frac{\Gamma, z_k{:}T \to T', x_i{:}T \vdash e{:}T' \quad \Gamma, z_k{:}T \to T' \vdash e'{:}T''}{\Gamma \vdash \textbf{letrec}\ z_k{:}T \to T' = \lambda x_i{:}T.e\ \textbf{in}\ e'{:}T''}$$

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

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

Figure 9: Dynamic Rebinding Calculus λ_{marsh}: Typing

type-T values, and rules

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

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

Partitioning the run-time errors into e err for the usual projection/application errors, together with unmarshalling of values not of the form **marshalled** Γ u, and e err' for the new errors above (defined in Figure 8), we have:

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

Theorem 6 (Type Preservation for λ_{marsh})
If $\vdash e\!:\!T$ and $e \longrightarrow e'$ then $\vdash e'\!:\!T$

Theorem 7 (Partial Safety for λ_{marsh})
If $\vdash e\!:\!T$ then $\neg(e$ $\mathsf{err})$.

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

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

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

3.5 Implementation

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

3.6 Adding Distributed Communication

We now extend λ_{marsh} just enough to show examples of the rebinding scenarios from §1, defining a $\lambda^{\text{io}}_{\text{marsh}}$ calculus. Some examples are given in Figures 11 and 12, with the syntax of the calculus in Figure 10.

$\lambda^{\text{io}}_{\text{marsh}}$: **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	$::=$	$z_i \mid n \mid () \mid (e, e') \mid \pi_r\ e \mid \lambda x_i{:}T.e \mid ee'$
		$\mid$	$\mathbf{let}\ z_k{:}T = e\ \mathbf{in}\ e' \mid \mathbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \mathbf{in}\ e'$
		$\mid$	$\mathbf{mark}\ M\ \mathbf{in}\ e \mid \mathbf{marshal}\ M\ e \mid \mathbf{marshalled}\ \Gamma\ u \mid \mathbf{unmarshal}\ M\ e$
		$\mid$	$\mathbf{ret}_T \mid c \mid e!e' \mid e?e' \mid s$
Configurations	P	$::=$	$0 \mid t{:}e \mid (P \mid P')$

Binding and alpha equivalence as in λ_{marsh}.

Figure 10: Distributed λ_{marsh}: $\lambda^{\text{io}}_{\text{marsh}}$ – Syntax

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

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

The semantics (given in Figure 13 and discussed in more detail below) defines a transition relation $P \xrightarrow{l} P'$ over configurations where the labels l are either empty, $t{:}f\ u$ for an invocation by thread t of library call $f{:}T \to T'$ from Γ_{lib}, with argument u, or $t{:}u$ for a return of value u from the OS to such an invocation. The (marshal) and (unmarshal) rules must be modified slightly to deal with external identifiers.

Communication between threads is by asynchronous message passing on typed channels c, with output and input forms $e!e'$ and $e?e'$. Only marshalled values should be communicated, so communications are typed as below (the full type system is in Figure 13).

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

Simple: $P =$

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

Secure encapsulation: $Q =$

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

Repeated rebinding: $R =$

$$\begin{array}{l} t_1{:}\ \textbf{let } here_0 = \text{“site 1” } \textbf{in} \\ \quad \textbf{mark } AppLib \textbf{ in} \\ \quad \textbf{letrec } f_0{:}\mathsf{unit} \to \mathsf{unit} = \\ \quad\quad \lambda x_0{:}\mathsf{unit}. \\ \quad\quad\quad \textbf{let } \_ = print_0 \text{“leaving: ” } \textbf{in} \\ \quad\quad\quad \textbf{let } \_ = print_0 here_0 \textbf{ in} \\ \quad\quad\quad c!\textbf{marshal } AppLib\ f_0 \textbf{ in} \\ \quad f_0() \\ \\ \mid t_2{:}\textbf{let } here_0 = \text{“site 2” } \textbf{in} \\ \quad \textbf{mark } AppLib \textbf{ in} \\ \quad c?(\lambda g_0{:}\mathsf{Marsh}\ (\mathsf{unit} \to \mathsf{unit}).(\textbf{unmarshal } AppLib\ g_0)()) \\ \\ \mid t_3{:}\textbf{let } here_0 = \text{“site 3” } \textbf{in} \\ \quad \textbf{mark } AppLib \textbf{ in} \\ \quad c?(\lambda g_0{:}\mathsf{Marsh}\ (\mathsf{unit} \to \mathsf{unit}).(\textbf{unmarshal } AppLib\ g_0)()) \end{array}$$

Figure 11: Dynamic Rebinding with IO and Communication: $\lambda^{\text{io}}_{\text{marsh}}$ Examples

Example P in Figure 11 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.

Our rebinding calculus is powerful enough to perform customized linking, useful for implementing secure encapsulation. Example Q is similar to P but the receiver defines two marks to be linked against, *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[2]. Which context to use

[2]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.

Moving Marks: S[=]

t_1: **let** $here_0$ = "site 1 – internal" **in**
mark *OuterLib* **in**
let x_0 = "internal resource (from site 1)" **in**
mark *InnerLib* **in**
let $send\_to\_external_0$:(unit → unit) → unit =
λz_0:(unit → unit).$c\_external$!**marshal** *OuterLib* z_0 **in**
let $send\_to\_internal_0$:(unit → unit) → unit =
λz_0:(unit → unit).$c\_internal$!**marshal** *InnerLib* z_0 **in**
let _ = ...use x_0... **in**
$send\_to\_external_0(\lambda y_0$:unit.
let _ = ...use x_0... **in**
let _ = $send\_to\_internal_0(\lambda w_0$:unit.
let _ = ...use x_0... **in**
()))

| t_2:**let** $here_0$ = "site 2 – external" **in**
mark *OuterLib* **in**
$c\_external?(\lambda g_0$:Marsh (unit → unit).(**unmarshal** *OuterLib* g_0)()

| t_3:**let** $here_0$ = "site 3 – internal" **in**
mark *OuterLib* **in**
let x_0 = "internal resource (from site 3)" **in**
mark *InnerLib* **in**
$c\_internal?(\lambda g_0$:Marsh (unit → unit).(**unmarshal** *InnerLib* g_0)()

Figure 12: Dynamic Rebinding with IO and Communication: Further $\lambda^{\text{io}}_{\text{marsh}}$ Examples

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

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

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 lambda-term, thus preventing a rebinding of *here* at the remote site.

In R, again in Figure 11, 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 12 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, 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.

Typing and Semantics The external library calls in Γ_{lib}, for example $\mathit{print}_0{:}\mathsf{string} \to \mathsf{unit}$, are invoked from within the language by application (rather than by a special system-call primitive). They therefore require no special typing treatment, though we do require that the types in $\text{ran}(\Gamma_{\text{lib}})$ are all of $T \to T'$ forms. The semantics has a ‘delta’ rule (lib-app) for invocations. On the right-hand-side of (lib-app) the application $(E_2.f_i)u$ is replaced by 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 [Sew97]). 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, indeed, one of a type not involving any function spaces). This is done in (lib-app) by the auxiliary

$$\begin{array}{lcl}
[\![n]\!] & = & n \\
[\![()]\!] & = & () \\
[\![(u, u')]\!] & = & ([\![u]\!], [\![u']\!]) \\
[\![\lambda x_i{:}T.e]\!] & = & \lambda x_i{:}T.e \\
[\![\mathbf{let}\ z_k{:}T = u\ \mathbf{in}\ u']\!] & = & \{[\![u]\!]/z_k\}[\![u']\!] \\
[\![\mathbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \mathbf{in}\ u]\!] & = & \{\lambda x_i{:}T.\mathbf{letrec}\ z_k{:}T' = \lambda x_i{:}T.e\ \mathbf{in}\ e/z_k\}[\![u]\!] \quad \text{if } z_k \neq x_i \\
[\![z_k]\!] & = & z_k \\
[\![\mathbf{mark}\ M\ \mathbf{in}\ u]\!] & = & \mathbf{mark}\ M\ \mathbf{in}\ [\![u]\!] \\
[\![\mathbf{marshalled}\ \Gamma\ u]\!] & = & \mathbf{marshalled}\ \Gamma\ u \\
[\![c]\!] & = & c \\
[\![s]\!] & = & s
\end{array}$$

generalising the analogous function from λ_d. Finally, the value returned from an external call must be well-typed. The side-condition $\Delta, \Gamma_{\text{lib}} \vdash u'{:}T'$ 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.

Turning to **marshal** and **unmarshal**, the rules are straightforward adaptions of the corresponding λ_{marsh} rules. In (marshal), note that we record Γ_{lib} in the grabbed 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 ordering, regarding it as a list of type assumptions $x_i{:}T$) to $\text{thb}(E_3)$ to calculate the appropriate rebinding

$\boxed{\Delta, \Gamma \vdash e{:}T}$

$$\frac{}{\Delta, \Gamma, x_i{:}T \vdash x_i{:}T}$$

$$\frac{}{\begin{array}{c}\Delta, \Gamma \vdash n{:}\mathsf{int}\\ \Delta, \Gamma \vdash (){:}\mathsf{unit}\\ \Delta, \Gamma \vdash s{:}\mathsf{string}\end{array}} \qquad \frac{\begin{array}{c}\Delta, \Gamma \vdash e{:}T\\ \Delta, \Gamma \vdash e'{:}T'\end{array}}{\Delta, \Gamma \vdash (e, e'){:}T * T'} \qquad \frac{\Delta, \Gamma \vdash e{:}T_1 * T_2}{\begin{array}{c}\Delta, \Gamma \vdash \pi_1\ e{:}T_1\\ \Delta, \Gamma \vdash \pi_2\ e{:}T_2\end{array}}$$

$$\frac{\Delta, \Gamma, x_i{:}T \vdash e{:}T'}{\Delta, \Gamma \vdash \lambda x_i{:}T.e{:}T \to T'} \qquad \frac{\begin{array}{c}\Delta, \Gamma \vdash e'{:}T \to T'\\ \Delta, \Gamma \vdash e{:}T\end{array}}{\Delta, \Gamma \vdash e'\,e{:}T'} \qquad \frac{\begin{array}{c}\Delta, \Gamma \vdash e{:}T\\ \Delta, \Gamma, x_i{:}T \vdash e'{:}T'\end{array}}{\Delta, \Gamma \vdash \textbf{let}\ x_i{:}T = e\ \textbf{in}\ e'{:}T'}$$

$$\frac{\begin{array}{c}\Delta, \Gamma, z_k{:}T \to T', x_i{:}T \vdash e{:}T'\\ \Delta, \Gamma, z_k{:}T \to T' \vdash e'{:}T''\end{array}}{\Delta, \Gamma \vdash \textbf{letrec}\ z_k{:}T \to T' = \lambda x_i{:}T.e\ \textbf{in}\ e'{:}T''}$$

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

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

$$\frac{}{\Delta, \Gamma \vdash \textbf{ret}_T : T} \qquad \frac{}{(\Delta, c{:}T), \Gamma \vdash c{:}T}$$

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

$\boxed{\Delta, \Gamma \vdash P\ \mathsf{ok}}$

$$\frac{}{\Delta, \Gamma \vdash 0\ \mathsf{ok}} \qquad \frac{\Delta, \Gamma \vdash e{:}\mathsf{unit}}{\Delta, \Gamma \vdash t{:}e\ \mathsf{ok}} \qquad \frac{\begin{array}{c}\Delta, \Gamma \vdash P\ \mathsf{ok}\\ \Delta, \Gamma \vdash P'\ \mathsf{ok}\\ \mathrm{tids}(P) \cap \mathrm{tids}(P') = \varnothing\end{array}}{\Delta, \Gamma \vdash P \mid P'\ \mathsf{ok}}$$

where $\mathrm{tids}(0) = \varnothing$, $\mathrm{tids}(t{:}e) = \{t\}$, and $\mathrm{tids}(P \mid P') = \mathrm{tids}(P) \cup \mathrm{tids}(P')$.

Figure 13: Distributed λ_{marsh}: $\lambda^{\text{io}}_{\text{marsh}}$ – Typing

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

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

Rules for invocations and returns of library calls:

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

The rule for communication:

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

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 $(P \mid P') \mid P'' \equiv P \mid (P' \mid P'')$.

Figure 14: Distributed λ_{marsh}: $\lambda^{\text{io}}_{\text{marsh}}$ – Semantics

substitution. 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.

For communication, typing ensures that channels only carry values of $\mathsf{Marsh}\ T$ types. These are always closed, so the (comm) rule for synchronisation can simply move them from sender to receiver. As a mild variant, one could insist that external library calls are of $(\mathsf{Marsh}\ T) \rightarrow (\mathsf{Marsh}\ T')$ types, obviating the need for $[\![\_]\!]$ in (lib-app) but requiring many more **marshal** and **unmarshal**s.

The values and evaluation contexts are very similar to those of λ_{marsh}. 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, grabbed value and lambda before (comm) fires.

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}.

3.7 Discussion

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

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

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

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

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

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

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

With one class one could specify the variables to be rebound either with marks or by explicitly annotating **marshal** with the set of rebindable identifiers. We believe the latter would be cumbersome in practice (with large sets of standard library identifiers). It would also be conceptually complex and

Simple Update Calculus: Syntax

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

Simple Update Calculus: Semantics

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

Figure 15: Simple Update Calculus: λ_{update}

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:

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

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.

4 Simple Update Calculus: λ_d + update

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

The λ_{update}-calculus is given in Figure 15 (the λ_d rules and error rules are elided). As in §3 it is convenient to use tagged identifiers and explicitly-typed **let**s, but the types are omitted in examples. We allow the programmer to place an expression **update** at points in the code where an update could occur; defining such updating 'safe points' is useful for ensuring programs behave properly [Hic01]. The intended semantics is that this expression will block, waiting for an update (possibly null) to be fed in.

An update can modify any identifier that is within its scope (at update-time), for example in

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

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

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

The static typing rule for **update** is trivial, as it is simply an expression of type unit. Naturally we have to perform some type checking at run-time; this is the second condition in the transition rule in Figure 15. Notice however, that we do not have to type-check the whole program; it suffices to check that the expression to be bound to the given identifier has the required type in the context that it will evaluate in. The other conditions of the transition rule are similarly straightforward. The first ensures that a rebinding substitution is defined, *i.e.* that the context E_3 has hole binders that are alpha-equivalent to the free variables of e. Here $\text{rebind}(V, L)$, for a set V and list L of variables, is defined if for all $x_i \in V$ there is some j with $x_j \in L$, in which case it is the the substitution taking each such x_i to the rightmost such x_j. The third condition ensures that the binding being updated, x_i, is the closest such binding occurrence for x (notice that an equivalence class x is specified for the update, but that the closest enclosing member, x_i, of this class is chosen as the updated binding). These conditions are sufficient to ensure that the following theorems hold. Their proofs are straightforward.

Theorem 8 (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; (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 9 (Type preservation for updates)
If $\vdash e{:}T$ and $e \xrightarrow{\{x \Leftarrow e'\}} e''$ then $\vdash e''{:}T$

Theorem 10 (Safety for updates)
If $\vdash e{:}T$ then $\neg(e\ \mathsf{err})$.

Our use of delayed instantiation cleanly supports updating higher-order functions. As we have mentioned before, this is a significant advance on previous treatments. Consider the following program:

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

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

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

5 Related Work

5.1 Lambda Calculi

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

In work on the compilation of extended recursion (particularly for mixin modules) Hirschowitz, Leroy, and Wells have (independently) used a semantics which is similar to λ_d save that (a) the language allows more general recursive definitions, and (b) the semantics collapses multiple **let**s [HLW03, Hir03]. It draws on work of Ariola and Blom [AB02] which also collapses **let** blocks. For rebinding, we need to preserve this structure.

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

5.2 Dynamic Rebinding and λ_{marsh}

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

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

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

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

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

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

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

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

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

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

Dynamic Linking Dynamic linking is a ubiquitous simple form of dynamic binding, allowing program bindings to be resolved either at load-time or run-time, rather than statically. Conventional executables will, when run, dynamically link shared libraries for standard library functions (*e.g.*, `read`, `write`, etc.). 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 [DE, dlo, $\mathrm{L}^{+}01$, Rou96, AVWW96]. Dynamic linking has been formally modelled for low-level machine code [Dug00, HWC00, HW00], and high-level languages like Java [DE]. Several authors have considered customised linking for security, performance, or debugging purposes [Rou96, SNC00, HWC00, SV00].

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

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

5.3 Dynamic Update

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

6 Conclusions and Future Work

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

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

There are several directions that are worth pursuing. Firstly, we would like a type system for λ_{marsh}that can statically prevent all run-time errors for programs that make only simple use of **marshal** and **unmarshal**. Whether this is possible without excessive complexity is unclear. The main difficulty seems to be capturing the ways in which the environment of a mark can change – one might speculate that an enriched term structure that explicitly records the DAG of scopes would enable a type preservation proof. Part of the motivation for this work is to cope with marshalling of values in distributed functional languages, but this paper does not deal with issues of type coherence between separately-compiled run-times. One might combine $\lambda^{\mathrm{io}}_{\mathrm{marsh}}$ with the hash types of [LPSW03].

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

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

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

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

A Proofs for λ_c, λ_r and λ_d: decomposition and typing

A.1 Unique redex/context decomposition

Theorem 11 (Unique redex/context decomposition for construct-time) *Let e be a closed expression. Then (in the construct-time calculus) exactly one of the following holds:*

1. *e is a value*
2. *e err*
3. *there exists a triple (E, e', rn) such that $E.e' = e$ and e' is an instance of the left-hand side of rule rn.*

Furthermore, if such a triple exists then it is unique.

Theorem 12 (Unique redex/context decomposition for redex-time) *Let e be an expression. Then (in the redex-time calculus) exactly one of the following holds:*

1. *e is a value ("e val").*
2. *there exists a pair (E_3, z) such that $E_3.z = e$ and z is a variable not contained in $hb(E_3)$ ("e var").*
3. *e err*
4. *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 ("e red").*

Furthermore, if such a pair or triple exists then it is unique.

Proof Observe firstly that e err implies $\exists E_3, e', r, u. e = E_3.(\pi_r\, e')$ or $e = E_3.(e' u)$, and that e err implies $E_3.e$ err for all E_3. Thus e err and e red are closed under general (E_3) context composition. e var is closed under E_1 context composition, since E_1 contexts are not binding and so do not affect the hole binders of the context. Observe further that any value u may be decomposed into a maximal binding context E_2 and an outermost-structure-manifest value w such that $E_2.w = u$.

The proof is by induction on the structure of e.

case z **:**

Let $E_3 = \_$. Then e var holds uniquely, and no other disjunct holds.

case n, $()$, or $\lambda z{:}T.e$ **:**

e val holds, and no other disjunct holds.

case (e_1, e_2) **:**

Observe that e itself is not a variable or the LHS of any rule; nor does it match any errors not in e_1 or e_2. Observe also that if e_1 is not a value, then (e_1, e_2) is not a value, and the only non-trivial decomposition is $(\_, e_2).e_1$.

Consider e_1. By I.H., there are four distinct cases:

case e_1 err **:**

e err. To prove no other disjunct holds, note we have already observed that e is not a value, that e is not itself a variable or the LHS of a rule, and by induction e_1 is not. But by our other observation above, these are the only possibilities; hence no other disjunct holds.

case e_1 red **:**
e red uniquely, and no other disjunct holds.

case e_1 var **:**
e var uniquely, and no other disjunct holds.

case e_1 val **:**
Observe that if e_1 is a value and e_2 is not, then (e_1, e_2) is not a value, and the only non-trivial decompositions of e are $(e_1, \_).e_2$ and $(\_, e_2).e_1$.
Consider e_2. By I.H., there are four distinct cases:

case e_2 err **:**
e err, and no other disjunct holds.

case e_2 red **:**
e red uniquely, and no other disjunct holds.

case e_2 var **:**
e var uniquely, and no other disjunct holds.

case e_2 val **:**
e val by definition of value, and no other disjunct holds.

case $\pi_r\, e_1$ **:**
Observe that the only non-trivial decomposition is $(\pi_r\, \_).e_1$; further, e is certainly itself not a value or a variable. Consider e_1. By I.H., there are four distinct cases:

case e_1 err **:**
e err, and no other disjunct holds.

case e_1 val **:**
Decompose e_1 into a maximal binding context E_2 and an outermost-structure-manifest value e_1'. If $e_1' = (u_1, u_2)$ then e red: e reduces by (proj), uniquely with $E_3 = \_$, $e' = e$, rn =proj, and no other disjunct holds. Otherwise, e err by (proj-err), and no other disjunct holds.

case e_1 var with (E_3', z) **:**
e var uniquely with $E_3 = (\pi_r\, \_).E_3'$; $\neg(e$ red$)$ because e_1 not (u_1, u_2), $\neg(e$ err$)$ because e_1 not a value, $\neg(e$ val$)$ by definition.

case e_1 red with (E_3', e_1', rn) **:**
e red with $E_3 = (\pi_r\, \_).E_3'$, $e' = e_1'$. Unique because the only other decomposition is irreducible, since e_1 not (u_1, u_2). No other disjunct holds.

case $e_1\, e_2$ **:**
Follows the same pattern as (e_1, e_2), but for the case e_1 val and e_2 val, decompose each into $E_2.u_1$, $E_2'.u_2$. Now if $u_1 = \lambda z{:}T.e_1'$, e red: uniquely, ensure by α-conversion that $\mathrm{fv}(e_2) \notin \mathrm{hb}(E_2)$ and reduce by (app) with $(\_, e, \mathrm{app})$. Otherwise, e err by (app-err). In each case, no other disjunct holds.

For the e_1 var and e_2 var cases, proceed as for $(\pi_r\, \_)$. No other disjunct holds, since a variable is not a value.

case let $z = e_1$ **in** e_2 **:**
There are three possible decompositions: (A): $\_.$**let** $z = e_1$ **in** e_2, (B): **let** $z = \_$ **in** $e_2.e_1$, (C): **let** $z = e_1$ **in** $\_.e_2$ if e_1 val. Consider e_1. By I.H., there are four distinct cases:

case e_1 err **:**
e err. (C) is not possible. No other disjunct can hold via (B). Cannot reduce by (A) since e_1 non-value; not a value; not a var.

case e_1 red with (E_3', e_1', rn) **:**
e red with $E_3 = \textbf{let}\ z = \_\ \textbf{in}\ e_2.E_3'$, $e' = e_1'$. (C) not possible; no other disjunct holds via (A) or (B).

case e_1 var with (E_3', z') **:**
e var with $E_3 = \textbf{let}\ z = \_\ \textbf{in}\ e_2.E_3'$. (C) not possible; no other disjunct holds via (A) or (B).

case e_1 val **:**
Observe now that (B) cannot yield e red or e var or e err. Consider e_2. By I.H., there are four distinct cases:

case e_2 err **:**
e err, and no other disjunct holds.

case e_2 red with (E_3', e_2', rn) **:**
e red with $E_3 = \textbf{let}\ z = e_1\ \textbf{in}\ \_.E_3'$, $e' = e_2'$. Since $\neg(e_2$ var$)$, this is unique. No other disjuncts hold.

case e_2 var with (E_3', z') **:**
If $z' \equiv z$, then we know by I.H. that $z' \notin \mathrm{hb}(E_3')$. Ensure $\mathrm{fv}(e_1) \notin \{z\} \cup \mathrm{hb}(E_3')$ by α-conversion. Then e red by $(\_, e, \mathrm{inst})$. Otherwise, $z' \notin \mathrm{hb}(E_3') \cup \{z\}$, so e var by $(\textbf{let}\ z = e_1\ \textbf{in}\ \_.E_3', z')$.

case e_2 val **:**
e val.

case letrec $z = \lambda x{:}T.e_1$ **in** e_2 **:**
Similar to **let** above. □

Theorem 13 (I.H. for Unique redex/context decomposition for destruct-time) *Let e be an expression. Then (in the destruct-time calculus) exactly one of the following holds:*

1. *e val: e is a value and $\neg(e$ var$_2)$ (value: may be benign unbound variable).*
2. *e var$_1$: there exist E_3, R, E_2, z such that $E_3.R.E_2.z = e$ and $z \notin hb(E_3.R.E_2)$ (unbound variable in destruct position).*
3. *e var$_2$: there exist E_2, z such that $E_2.z = e$ and $z \in hb(E_2)$ (value: bound variable).*
4. *e err$_1$: e err and $\neg(e$ var$_1)$ (fatal error).*
5. *e red: 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 (reducible).*

Furthermore, if such a pair, triple, or quadruple exists then it is unique.

Note that "e a value" means "e val or e var$_2$".

Proof Observe firstly that e err$_1$ implies e err implies $\exists E_3, e_0, R.e = E_3.R.e_0$, and that e err$_1$ implies $E_3.e$ err$_1$ for all E_3. Thus e var$_2$, e err$_1$, and e red are all closed under general (E_3) context composition. e var$_1$ is closed under E_1 context composition, since E_1 contexts are not binding and so do not affect the hole binders of the context. Observe further that any value u may be decomposed into a maximal binding context E_2 and an outermost-struture-manifest value w such that $E_2.w = u$.

The proof is by induction on the structure of e.

case z **:**
e val, and no other disjunct holds.

case n, $()$, or $\lambda z{:}T.e$ **:**

e val, and no other disjunct holds.

case (e', e'') **:**

Observe that e is not var_2, and is not itself the LHS of any rule; nor does e match any errors not in e' or e''. Observe also that if e' is not a value, then (e', e'') is not a value, and the only non-trivial decomposition is $(\_, e'').e'$; then since $(\_, e'') \notin R$, $\neg(e'\ \mathsf{var}_1) \implies \neg(e\ \mathsf{var}_2)$.

Consider e'. By I.H., there are five distinct cases:

case e' err_1 **:**

e err_1. To prove no other disjunct holds, note we have already observed that e is not a value or var_1, that e is not itself a variable or the LHS of a rule, and by induction e' is not. But by our other observation above, these are the only possibilities; hence no other disjunct holds.

case e' red **:**

e red uniquely, and no other disjunct holds.

case e' var_1 **:**

e var_1 uniquely, and no other disjunct holds.

case e' var_2 or e' val **:**

Observe that if e' is a value and e'' is not, then (e', e'') is not a value, and the only non-trivial decompositions of e are $(e', \_).e''$ and $(\_, e'').e'$.

Consider e''. By I.H., there are five distinct cases:

case e'' err_1 **:**

e err_1, and no other disjunct holds.

case e'' red **:**

e red uniquely, and no other disjunct holds.

case e'' var_1 **:**

e var_1 uniquely, and no other disjunct holds.

case e'' var_2 or e'' val **:**

e val by definition of value, and no other disjunct holds.

case $\pi_r\ e'$ **:**

Observe that the only non-trivial decomposition is $(\pi_r\ \_).e'$; further, e is certainly itself not a value or var_2, and $(\pi_r\ \_)$ is not a binding context. Consider e'. By I.H., there are five distinct cases:

case e' err_1 **:**

e err_1, and no other disjunct holds.

case e' val **:**

Decompose e' into a maximal binding context E_2 and an outermost-structure-manifest value e'_0. If $e'_0 = (u_1, u_2)$ then e red: e reduces by (proj), uniquely with $E_3 = \_$, $e_0 = e = \pi_r\ (E_2.e'_0)$, rn =proj, and no other disjunct holds. If $e'_0 = z$ then we know $z \notin \mathrm{hb}(E_2)$ (for if not, e' var_2), and so e var_1, since $(\pi_r\ \_) \in R$, and $\neg(e\ \mathsf{err}_1)$ by definition, and no other disjunct holds. Otherwise, e err_1 by (proj-err), and no other disjunct holds.

case e' var_1 **:**

e var_1, and no other disjunct holds.

case e' var_2 with (E'_2, z) **:**

e red with $E_3 = \_$, $e_0 = e = \pi_r\ (E'_2.z)$, rn =inst-2 or instrec-2 (according to whether z is bound in E'_2 by a **let** or a **letrec**). $\neg(e\ \mathsf{err}_1)$ and $\neg(e\ \mathsf{var}_1)$ since $z \in \mathrm{hb}(E'_2)$.

case e' red with (E'_3, e'_0, rn) **:**
e red with $E_3 = (\pi_r\,\_).E'_3$, $e_0 = e'_0$. Unique because the only other decomposition is irreducible, since e' not (u_1, u_2). No other disjunct holds.

case $e'e''$ **:**

Possible decompositions are $\_.(e'e'')$, or $(\_e'').e'$ (which is an R context if e'' is a value), or $(e'\_).e''$ if e' a value. Observe that e is not var$_2$ or value, and neither decomposition involves a binding context.

Consider e'. By I.H., there are five distinct cases:

case e' err$_1$ **:**
e err$_1$, and no other disjunct holds.

case e' red **:**
e red uniquely, and no other disjunct holds.

case e' var$_1$ **:**
e var$_1$ uniquely (since $(\_e'')$ not a binding context), and no other disjunct holds (since e' not a value).

case e' var$_2$ with (E'_2, z) **:**
Now e' is a value, and so we must consider e''. By I.H., there are five distinct cases:

case e'' err$_1$ **:**
e err$_1$, and no other disjunct holds.

case e'' red **:**
e red uniquely, and no other disjunct holds.

case e'' var$_1$ **:**
e var$_1$ uniquely, and no other disjunct holds (since e'' not value).

case e'' var$_2$ or e'' val **:**
e red with $E_3 = \_$, $e_0 = e = (\_e'').E'_2.z$, rn =inst-2 or instrec-2 (according to whether z is bound in E'_2 by a **let** or a **letrec**). $\neg(e$ err$_1)$ and $\neg(e$ var$_1)$ since $z \in \text{hb}(E'_2)$, no other reduction or error because e' and e'' both values, and $e' \neq E'_2.\lambda z{:}T.e'''$.

case e' val **:**
Decompose e' into $E'_2.e'_0$.
Consider e''. By I.H., there are five distinct cases:

case e'' err$_1$ **:**
e err$_1$, and no other disjunct holds.

case e'' red **:**
e red uniquely, and no other disjunct holds.

case e'' var$_1$ **:**
e var$_1$ uniquely, and no other disjunct holds (since e'' not value).

case e'' var$_2$ or e'' val **:**
If $e'_0 = \lambda z{:}T.e'''$ then e red with $E_3 = \_$, $e_0 = e = ((E'_2.\lambda z{:}T.e''')e'')$, rn =app (ensuring $\text{fv}(e'') \notin E'_2$ by α-conversion), and no other disjunct holds. If $e'_0 = z$ then we know $z \notin \text{hb}(E'_2)$ (for if not, e' var$_2$), and so e var$_1$, since $(\_e'') \in R$ and and $\neg(e$ err$_1)$ by definition, and no other disjunct holds. Otherwise, e err$_1$ by (app-err), and no other disjunct holds.

case let $z = e'$ **in** e'' **:**

There are three possible decompositions: $\_.$**let** $z = e'$ **in** e'', **let** $z = \_$ **in** $e''.e'$, **let** $z = e'$ **in** $\_.e''$ if e' a value.

Consider e'. By I.H., there are five distinct cases:

case e' err_1 **:**
e' err_1, and no other disjunct holds.

case e' red with (E'_3, e'_0, rn) **:**
e red with $E_3 = \textbf{let}\ z = \_\ \textbf{in}\ e''.E'_3$, $e_0 = e'_0$, uniquely, and no other disjunct holds.

case e' var_1 **:**
then e var_1 uniquely, and no other disjunct holds.

case e' var_2 or e' val **:**
Consider e''. By I.H., there are five distinct cases:

case e'' err_1 **:**
e err_1, and no other disjunct holds.

case e'' red with (E'_3, e''', rn) **:**
e red with $E_3 = \textbf{let}\ z = e'\ \textbf{in}\ \_.E'_3$, $e_0 = e'''$. This is unique. No other disjuncts hold.

case e'' var_2 **:**
e var_2, and no other disjunct holds.

case e'' val **:**
Decompose e'' into $E''_2.u''$. If $u'' = z$, where z is the bound variable of the **let** , then e var_2, uniquely, and no other disjunct holds. Otherwise, e val, and no other disjunct holds.

case e'' var_1 with $e'' = E''_3.R''.E''_2.z''$ and $z'' \notin \mathrm{hb}(E''_3.R''.E''_2)$ **:**
If $z'' \equiv z$, then we know that $z \notin \mathrm{hb}(E''_3, E''_2)$. Ensure $\mathrm{fv}(e') \notin \{z\} \cup \mathrm{hb}(E''_3, E''_2)$ by α-conversion. Then e red with $E_3 = \_$, $e_0 = e = \textbf{let}\ z = e'\ \textbf{in}\ E''_3.R''.E''_2.z$, rn =inst-1. Otherwise, e var_1, since $z'' \notin \{z\} \cup \mathrm{hb}(E''_3, E''_2)$.

case letrec $z = \lambda x{:}T.e'$ **in** e'' **:**
Similar to **let** above, but note the different scope of z. □

Observe that at the top level e $\mathsf{var}_1 \implies e$ err, and e $\mathsf{var}_2 \implies e$ a value. Hence:

Corollary 14 (Unique redex/context decomposition for destruct-time) *Let e be an expression. Then (in the destruct-time calculus) exactly one of the following holds:*

1. *e is a value.*
2. *e err.*
3. *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.*

Furthermore, if such a triple exists then it is unique.

A.2 Type preservation and safety

Lemma 15 (Renaming) *If $\Gamma \vdash e{:}T$ and Γ', e' are obtained from Γ, e by an injective renaming of $dom(\Gamma) \cup fv(e)$ then $\Gamma' \vdash e'{:}T$.*

Lemma 16 (Weakening) *If $\Gamma, \Gamma'' \vdash e{:}T$ and $dom((\Gamma, \Gamma'')) \cap dom(\Gamma') = \{\}$ then $\Gamma, \Gamma', \Gamma'' \vdash e{:}T$.*

Proof Induction on type derivations, using Lemma 15 in the lambda and let cases. □

Lemma 17 (Permutation) *If $\Gamma, \Gamma', \Gamma'', \Gamma''' \vdash e{:}T$ then $\Gamma, \Gamma'', \Gamma', \Gamma''' \vdash e{:}T$.*

Proof Induction on type derivations. □

It is convenient to work with E_2 and E_3 contexts in which the hole binders are distinct from each other and from a set of identifiers. Accordingly, we define a predicate $\mathrm{dhb}(E_2, X)$ as the least such that

- $\mathrm{dhb}(\_, X)$
- $\mathrm{dhb}(E_2, X) \wedge z \notin \mathrm{hb}(E_2) \cup X \implies \mathrm{dhb}(E_2.\mathbf{let}\ z = u\ \mathbf{in}\ \_, X)$
- $\mathrm{dhb}(E_2, X) \wedge z \notin \mathrm{hb}(E_2) \cup X \implies \mathrm{dhb}(E_2.\mathbf{letrec}\ z = \lambda x{:}T.e\ \mathbf{in}\ \_, X)$

and $\mathrm{dhb}(E_3, X)$ by similar clauses together with $\mathrm{dhb}(E_3, X) \implies \mathrm{dhb}(E_3.A_1, X)$. Strictly there are different definitions for λ_r and λ_d, as u ranges over different terms in each.

Lemma 18 (E_2 inversion for λ_r and λ_d) *If $\Gamma \vdash E_2.e{:}T$ and $dhb(E_2, dom(\Gamma))$ then there exists Γ' such that $\Gamma, \Gamma' \vdash e{:}T$, $dom(\Gamma') = hb(E_2)$, and $\forall e', T'. \Gamma, \Gamma' \vdash e'{:}T' \implies \Gamma \vdash E_2.e'{:}T'$.*

Proof The proofs for λ_r and λ_d are identical.

Induction on E_2.

Case $\_$. Trivial.

Case $E_2.(\mathbf{let}\ z = u\ \mathbf{in}\ \_)$. Suppose $\Gamma \vdash E_2.(\mathbf{let}\ z = u\ \mathbf{in}\ ).e{:}T$ and $\mathrm{dhb}(E_2.(\mathbf{let}\ z = u\ \mathbf{in}\ \_).e, \mathrm{dom}(\Gamma))$.
By defn dhb we have $\mathrm{dhb}(E_2, \mathrm{dom}(\Gamma))$ and $z \notin \mathrm{hb}(E_2) \cup \mathrm{dom}(\Gamma)$.
By ind.hyp. there exists Γ' such that $\Gamma, \Gamma' \vdash (\mathbf{let}\ z = u\ \mathbf{in}\ \_).e{:}T$, $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_2)$, and $\forall e', T'. \Gamma, \Gamma' \vdash e'{:}T' \implies \Gamma \vdash E_2.e'{:}T'$ (*).
By inversion of the typing relation there exist $\hat{z}, \hat{e}, T''$ such that $(\mathbf{let}\ z = u\ \mathbf{in}\ e) = (\mathbf{let}\ \hat{z} = u\ \mathbf{in}\ \hat{e})$, $\Gamma, \Gamma' \vdash u{:}T''$, and $\Gamma, \Gamma', \hat{z}{:}T'' \vdash \hat{e}{:}T$.
By Lemma 15 (as $z \notin \mathrm{hb}(E_2) \cup \mathrm{dom}(\Gamma) = \mathrm{dom}((\Gamma, \Gamma'))$) we have $\Gamma, \Gamma', z{:}T'' \vdash e{:}T$.
Trivially $\mathrm{dom}((\Gamma', z{:}T'')) = \mathrm{hb}(E_2.(\mathbf{let}\ z = u\ \mathbf{in}\ \_))$.
Now suppose $\Gamma, \Gamma', z{:}T'' \vdash e'{:}T'$.
By typing $\Gamma, \Gamma' \vdash (\mathbf{let}\ z = u\ \mathbf{in}\ \_).e'{:}T'$.
By (*) $\Gamma \vdash E_2.(\mathbf{let}\ z = u\ \mathbf{in}\ \_).e'{:}T'$.

Case $E_2.(\mathbf{letrec}\ z = \lambda x{:}T.\_)$. Similar.

□

Lemma 19 (E_3 inversion for λ_r and λ_d) *If $\Gamma \vdash E_3.e{:}T$ and $dhb(E_3, dom(\Gamma))$ then there exist Γ', T' such that $\Gamma, \Gamma' \vdash e{:}T'$, $dom(\Gamma') = hb(E_3)$, and $\forall e'.\Gamma, \Gamma' \vdash e'{:}T' \implies \Gamma \vdash E_3.e'{:}T$.*

Proof Induction on E_3.

For λ_r:

Case _. Trivial.

Case $E_3.A_1$

Case $E_3.(\_, e'')$. Suppose $\Gamma \vdash E_3.(\_, e'').e{:}T$ and $\mathrm{dhb}(E_3.(\_, e''), \mathrm{dom}(\Gamma))$.
By defn. dhb have $\mathrm{dhb}(E_3, \mathrm{dom}(\Gamma))$.
By ind.hyp. exist Γ', T' such that $\Gamma, \Gamma' \vdash (\_, e'').e{:}T'$, $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_3)$, and $\forall e'.\Gamma, \Gamma' \vdash e'{:}T' \implies \Gamma \vdash E_3.e'{:}T$ (*).
By inversion of the typing relation exist T_1', T_2' such that $T' = T_1' * T_2'$, $\Gamma, \Gamma' \vdash e{:}T_1'$, and $\Gamma, \Gamma' \vdash e''{:}T_2'$.
Trivially $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_3.(\_, e''))$.
Now suppose for some e' that $\Gamma, \Gamma' \vdash e'{:}T_1'$.
By typing $\Gamma, \Gamma' \vdash (\_, e'').e'{:}T_1' * T_2'$.
By (*) $\Gamma \vdash E_3.(\_, e'').e'{:}T$.

Cases $E_3.(u, \_)$, $E_3.(\pi_r\, \_)$, $E_3.(\_e'')$, $E_3.(u\_)$, $E_3.(\mathbf{let}\ z = \_\ \mathbf{in}\ e'')$. All similar.

Case $E_3.A_2$

Case $E_3.(\mathbf{let}\ z = u\ \mathbf{in}\ \_)$. Suppose $\Gamma \vdash E_3.(\mathbf{let}\ z = u\ \mathbf{in}\ \_).e{:}T$ and $\mathrm{dhb}(E_3.(\mathbf{let}\ z = u\ \mathbf{in}\ \_), \mathrm{dom}(\Gamma))$.
By defn dhb we have $\mathrm{dhb}(E_3, \mathrm{dom}(\Gamma))$ and $z \notin \mathrm{hb}(E_3) \cup \mathrm{dom}(\Gamma)$.
By ind.hyp. exist Γ', T' such that $\Gamma, \Gamma' \vdash (\mathbf{let}\ z = u\ \mathbf{in}\ \_).e{:}T'$, $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_3)$, and $\forall e'.\Gamma, \Gamma' \vdash e'{:}T' \implies \Gamma \vdash E_3.e'{:}T$ (*).
By inversion of the typing relation exist $\hat{z}, \hat{e}, T''$ such that $(\mathbf{let}\ z = u\ \mathbf{in}\ e) = (\mathbf{let}\ \hat{z} = u\ \mathbf{in}\ \hat{e})$, $\Gamma, \Gamma' \vdash u{:}T''$, and $\Gamma, \Gamma', \hat{z}{:}T'' \vdash \hat{e}{:}T'$.
By Lemma 15 (as $z \notin \mathrm{hb}(E_3) \cup \mathrm{dom}(\Gamma) = \mathrm{dom}((\Gamma, \Gamma'))$) we have $\Gamma, \Gamma', z{:}T'' \vdash e{:}T'$.
Trivially $\mathrm{dom}(\Gamma')z{:}T'' = \mathrm{hb}(E_3.(\mathbf{let}\ z = u\ \mathbf{in}\ \_))$.
Now suppose for some e' that $\Gamma, \Gamma', z{:}T'' \vdash e'{:}T'$.
By typing $\Gamma, \Gamma' \vdash (\mathbf{let}\ z = u\ \mathbf{in}\ \_).e'{:}T'$.
By (*) $\Gamma \vdash E_3.(\mathbf{let}\ z = u\ \mathbf{in}\ \_).e'{:}T$.

Case $E_3.(\mathbf{letrec}\ z = \lambda x{:}T.e''\ \mathbf{in}\ \_)$. Similar.

For λ_d: the proof is the same except for the different u.

□

Theorem 20 (Type preservation for λ_c, λ_r and λ_d) *If $\Gamma \vdash e{:}T$ and $e \longrightarrow e'$ then $\Gamma \vdash e'{:}T$.*

Proof For λ_c this is completely standard.

For λ_r we proceed by induction on derivations of $e \longrightarrow e'$.

Case (proj). Suppose $\Gamma \vdash \pi_r\,(E_2.(u_1, u_2)){:}T$.

W.l.g. take E_2, u_1, u_2 such that $\mathrm{dhb}(E_2, \mathrm{dom}(\Gamma))$.

By inversion of the typing relation there exist T_1, T_2 such that $T = T_r$ and $\Gamma \vdash E_2.(u_1, u_2){:}T_1 * T_2$.

By Lemma 18 there exists Γ' such that $\Gamma, \Gamma' \vdash (u_1, u_2){:}T_1 * T_2$, $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_2)$, and $\forall e', T'.\Gamma, \Gamma' \vdash e'{:}T' \implies \Gamma \vdash E_2.e'{:}T'$ (*).

By inversion of the typing relation $\Gamma, \Gamma' \vdash u_r{:}T_r$.

By (*) $\Gamma \vdash E_2.u_r{:}T$.

Case (app). Suppose $\Gamma \vdash (E_2.(\lambda z{:}T_0.e))u{:}T$.

W.l.g. take E_2, z, e such that $z \notin \mathrm{dom}(\Gamma)$ and $\mathrm{dhb}(E_2, \mathrm{dom}(\Gamma) \cup \{z\})$.

Aside: what does this 'without loss of generality' really mean? This: given the 7-tuple $\Gamma, E_2, z, T_0, e, u, T$ appearing in the typing hypothesis or in the premise or conclusion of the (app) rule, such that $\mathrm{fv}(u) \notin \mathrm{hb}(E_2)$ (from the sidecondition of (app)) then there exist $\hat{E_2}, \hat{z}, \hat{e}$ such that $\hat{z} \notin \mathrm{dom}(\Gamma)$, $\mathrm{dhb}(\hat{E_2}, \mathrm{dom}(\Gamma) \cup \{\hat{z}\})$, $\mathrm{fv}(u) \notin \mathrm{hb}(\hat{E_2})$, $(E_2.(\lambda z{:}T.e))u = (\hat{E_2}.(\lambda\hat{z}{:}T.\hat{e}))u$ and $(E_2.\mathbf{let}\ z = u\ \mathbf{in}\ e) = (\hat{E_2}.\mathbf{let}\ \hat{z} = u\ \mathbf{in}\ \hat{e})$ (the latter two being the terms of the typing hypothesis and (app) rule).

By inversion of the typing relation there exists T_1 such that $\Gamma \vdash E_2.(\lambda z{:}T_0.e){:}T_1 \to T$ and $\Gamma \vdash u{:}T_1$.

By Lemma 18 there exists Γ' such that $\Gamma, \Gamma' \vdash \lambda z{:}T_0.e{:}T_1 \to T$, $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_2)$, and $\forall e', T'.\Gamma, \Gamma' \vdash e'{:}T' \implies \Gamma \vdash E_2.e'{:}T'$ (*).

By inversion of the typing relation and by renaming $\Gamma, \Gamma', z{:}T_0 \vdash e{:}T$ and $T_0 = T_1$.

By weakening (Lemma 17) $\Gamma, \Gamma' \vdash u{:}T_0$.

By typing $\Gamma, \Gamma' \vdash \mathbf{let}\ z = u\ \mathbf{in}\ e{:}T$.

By (*) $\Gamma \vdash E_2.\mathbf{let}\ z = u\ \mathbf{in}\ e{:}T$.

Case (inst). Suppose $\Gamma \vdash \mathbf{let}\ z = u\ \mathbf{in}\ E_3.z{:}T$.

W.l.g. take E_3, z such that $z \notin \mathrm{dom}(\Gamma)$ and $\mathrm{dhb}(E_3, \mathrm{dom}((\Gamma, z{:}T_1)))$.

By inversion of the typing relation and by renaming there exists T_1 such that $\Gamma \vdash u{:}T_1$ and $\Gamma, z{:}T_1 \vdash E_3.z{:}T$.

By Lemma 19 there exist Γ', T' such that $\Gamma, z{:}T_1, \Gamma' \vdash z{:}T'$, $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_3)$, and $\forall e'.\Gamma, z{:}T_1, \Gamma' \vdash e'{:}T' \implies \Gamma, z{:}T_1 \vdash E_3.e'{:}T$ (*).

By inversion of the typing relation $T_1 = T'$.

By weakening (Lemma 17) $\Gamma, z{:}T_1, \Gamma' \vdash u{:}T_1$.

By (*) $\Gamma, z{:}T_1 \vdash E_3.u{:}T$.

By typing $\Gamma \vdash \mathbf{let}\ z = u\ \mathbf{in}\ E_3.u{:}T$.

Case (instrec). Consider the reduction $\mathbf{letrec}\ z = \lambda x{:}T_1.e\ \mathbf{in}\ E_3.z \longrightarrow \mathbf{letrec}\ z = \lambda x{:}T_1.e\ \mathbf{in}\ E_3.\lambda x{:}T_1.e$ with $z \neq x$ and $z \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(\lambda x{:}T_1.e) \notin \mathrm{hb}(E_3)$. Suppose $\Gamma \vdash \mathbf{letrec}\ z = \lambda x{:}T_1.e\ \mathbf{in}\ E_3.z{:}T$.

W.l.g. take E_3, e, z, x such that $z, x \notin \mathrm{dom}(\Gamma)$ and $\mathrm{dhb}(E_3, \mathrm{dom}(\Gamma) \cup \{z, x\})$.

By inversion of the typing relation and by renaming there exists T_2 such that $\Gamma, z{:}T_1 \to T_2, x{:}T_1 \vdash e{:}T_2$ and $\Gamma, z{:}T_1 \to T_2 \vdash E_3.z{:}T$.

By the above name assumption we have $\mathrm{dhb}(E_3, \mathrm{dom}(\Gamma) z{:}T_1 \to T_2)$.
By Lemma 19 there exist Γ', T' such that $\Gamma, z{:}T_1 \to T_2, \Gamma' \vdash z{:}T'$, $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_3)$, and $\forall e'.\Gamma, z{:}T_1 \to T_2, \Gamma' \vdash e'{:}T' \implies \Gamma, z{:}T_1 \vdash E_3.e'{:}T$ (*).
By inversion of the typing relation $T' = T_1 \to T_2$.
By typing $\Gamma, z{:}T_1 \to T_2 \vdash \lambda x{:}T_1.e{:}T_1 \to T_2$.
By weakening (Lemma 17) $\Gamma, z{:}T_1 \to T_2, \Gamma' \vdash \lambda x{:}T_1.e{:}T_1 \to T_2$.
By (*) $\Gamma, z{:}T_1 \vdash E_3.\lambda x{:}T_1.e{:}T$.
By typing $\Gamma \vdash \textbf{letrec}\ z = \lambda x{:}T_1.e\ \textbf{in}\ E_3.\lambda x{:}T_1.e{:}T$.

Case (E_3). By Lemma 19.

For λ_d: (proj) and (app) are as in λ_r. For (inst-1) and (inst-2) note that the destruct contexts R are contained in A_1, so both are E_3-closure instances of the λ_r (inst) rule (modulo the different u notions). The same proof therefore goes through. For the (instrec-1) and (instrec-2) cases the same applies.

□

Theorem 21 (Type safety for λ_c, λ_r and λ_d) *If* $\vdash e{:}T$ *then* $\neg(e\ \mathsf{err})$.

Proof Again, λ_c is standard.

Note that for λ_r a more general result holds, with an arbitrary Γ, but for λ_d it is important that the type context be empty.

For λ_r:

Case (proj-err). Suppose $\Gamma \vdash E_3.\pi_r\,(E_2.w){:}T$ and $\neg\exists u_1, u_2.w = (u_1, u_2)$ (*).
W.l.g. $\mathrm{dhb}(E_3, \mathrm{dom}(\Gamma))$ and $\mathrm{dhb}(E_2, \mathrm{dom}(\Gamma) \cup \mathrm{hb}(E_3))$.
By Lemma 19 there exist Γ', T' such that $\Gamma, \Gamma' \vdash \pi_r\,(E_2.w){:}T'$ and $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_3)$.
By inversion of the typing relation there exist T_1, T_2 such that $T' = T_r$ and $\Gamma, \Gamma' \vdash E_2.w{:}T_1 * T_2$.
Trivially $\mathrm{dhb}(E_2, \mathrm{dom}(\Gamma)\Gamma')$ so by Lemma 18 there exists Γ'' such that $\Gamma, \Gamma', \Gamma'' \vdash w{:}T_1 * T_2$.
The only w form which is typable with a product type is (u_1, u_2), contradicting (*).

Case (app-err). Similar.

For λ_d:

Case (proj-err). Suppose $\vdash E_3.\pi_r\,(E_2.w){:}T$ and $\neg\exists u_1, u_2.w = (u_1, u_2)$ (*) and $\neg\exists z\ \textbf{in}\ \mathrm{hb}(E_3, E_2).w = z$ (**).
W.l.g. $\mathrm{dhb}(E_3, \{\})$ and $\mathrm{dhb}(E_2, \mathrm{hb}(E_3))$.
By Lemma 19 there exist Γ', T' such that $\Gamma' \vdash \pi_r\,(E_2.w){:}T'$ and $\mathrm{dom}(\Gamma') = \mathrm{hb}(E_3)$.
By inversion of the typing relation there exist T_1, T_2 such that $T' = T_r$ and $\Gamma' \vdash E_2.w{:}T_1 * T_2$.
Trivially $\mathrm{dhb}(E_2, \mathrm{dom}(\Gamma'))$ so by Lemma 18 there exists Γ'' such that $\Gamma', \Gamma'' \vdash w{:}T_1 * T_2$.
The only w forms which are typable with a product type is (u_1, u_2) and z. The former contradicts (*). For the latter, by (**) z is free in $E_3.\pi_r\,(E_2.w)$, which contradicts its typability in the empty context.

Case (app-err). Similar.

□

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

Figure 16: Annotated syntax λ'

B Proofs for λ_c, λ_r and λ_d: Observational equivalence

Throughout this appendix we work with a simpler language, replacing **letrec** by a nonterminating Ω, with $\Omega \longrightarrow \Omega$ in all reduction strategies.

Theorem 22 (Observational Equivalence)

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

B.1 Observational equivalence between λ_r and λ_c

Theorem 23 states the sense in which we shall consider λ_c and λ_r to be observationally equivalent. We show the validity of this theorem using relational reasoning to establish the existence, by construction, of a weak bisimulation between the transition systems of λ_c and λ_r, furthermore we show that this relation preserves termination. For technical reasons the proof proceeds by introducing an intermediate language, $\lambda_{r'}$, given in figures 16 and 17, and then constructing a relation from λ_c to $\lambda_{r'}$ and another from $\lambda_{r'}$ to λ_r. Properties of these relations are then established for the sole purpose of proving that their composition is the required relation between λ_r and λ_c.

Theorem 23 *For all* $e \in \lambda$ *the following hold:*

1. $\vdash e{:}\mathsf{int} \implies (e \;\longrightarrow_c^*\; n \implies \exists v.\; e \;\longrightarrow_r^*\; u \;\wedge\; n = \{|\; v \;|\})$
2. $\vdash e{:}\mathsf{int} \implies (e \;\longrightarrow_r^*\; v \implies \exists n.\; e \;\longrightarrow_c^*\; n \;\wedge\; n = \{|\; v \;|\})$

We begin by making definitions of auxiliary functions and certain normal forms of expressions. Basic properties of these, that will be needed in the construction of the bisimulation, are then established.

Definition 1 (Environment)

An *environment* Φ is a list containing pairs whose first component is an identifier and whose second component is a c-value or an identifier that is the same as the first component. Environments have the property that $\forall x \in \mathrm{dom}(\Phi).\; \Phi(x) = v \;\wedge\; \forall z \in \mathrm{fv}(v).z {\leq_\Phi} x$ where $\leq_\Phi$ is the ordering of the identifiers in Φ. In addition we require that all the first components of the pairs in the list are disjoint. We write $\Phi, z \mapsto v$ for the disjoint extension of Φ forming a new environment. We write $\Phi[z \mapsto v]$ for the environment acting as Φ, but mapping z to v

When extending an environment, we must make sure that the free variables of the element we add are already contained in the environment. In practice this constraint is easily satisfied as the values added to the environment are of the from $[\![\; u \;]\!]^\Phi$ ($[\![\; - \;]\!]^-$ is defined in figure 18) where we know $\mathrm{fv}(u) \subseteq \mathrm{dom}(\Phi)$.

Reduction contexts

Values	u	$::=$	$n \mid () \mid (u, u') \mid \lambda x{:}T.a \mid \mathbf{let}_0\ z{:}T = u\ \mathbf{in}\ u$
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	$::=$	$\mathbf{let}_0\ z{:}T = u\ \mathbf{in}\ \_$
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 $\mathrm{fv}(u) \notin \mathrm{hb}(E_2)$	$\longrightarrow$	$E_2.\mathbf{let}_0\ x = u\ \mathbf{in}\ a$
(omega)	Ω	$\longrightarrow$	Ω
(inst)	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.z$ if $z \notin \mathrm{hb}(E_3)$ and $\mathrm{fv}(u) \notin z, \mathrm{hb}(E_3)$	$\longrightarrow$	$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.u$
(zero)	$\mathbf{let}_1\ z = u\ \mathbf{in}\ a$	$\longrightarrow$	$\mathbf{let}_0\ z = u\ \mathbf{in}\ a$

$$\text{(cong)}\quad \frac{a \longrightarrow a'}{E_3.a \longrightarrow E_3.a'}$$

Figure 17: $\lambda_{r'}$ calculus

It is easy to check that $\mathrm{fv}(u) \subseteq \mathrm{dom}(\Phi) \implies \mathrm{fv}(\llbracket u \rrbracket^{\Phi}) \notin \mathrm{dom}(\Phi)$, which guarantees that our extensions are safe.

Definition 2 (Well-formedness) We write $\mathrm{wf}[a]$ to denote that a term a is well-formed, in the sense of the definition below. It is parametrised on a value predicate val, a predicate determining if a given expression is a value in the calculus under consideration, the actual value of which should be clear from the context.

The definition uses an auxiliary predicate $\mathrm{nozeros}(a)$ which asserts that there are no subexpressions of a of the form $\mathbf{let}_0\ z = a\ \mathbf{in}\ a'$.

$$
\begin{array}{lcl}
\mathrm{wf}[z] & = & \mathrm{t} \\
\mathrm{wf}[n] & = & \mathrm{t} \\
\mathrm{wf}[()] & = & \mathrm{t} \\
\mathrm{wf}[\Omega] & = & \mathrm{t} \\
\mathrm{wf}[(a, a')] & = & \mathrm{wf}[a] \wedge \mathrm{wf}[a'] \\
\mathrm{wf}[\pi_r\ a] & = & \mathrm{wf}[a] \\
\mathrm{wf}[\lambda x{:}T.a]) & = & \mathrm{wf}[a] \wedge \mathrm{nozeros}(a) \\
\mathrm{wf}[a\ a'] & = & \mathrm{wf}[a] \wedge \mathrm{wf}[a'] \\
\mathrm{wf}[\mathbf{let}_0\ z = a\ \mathbf{in}\ a'] & = & \mathrm{wf}[a] \wedge \mathrm{wf}[a'] \wedge a\ \mathsf{val} \\
\mathrm{wf}[\mathbf{let}_1\ z = a\ \mathbf{in}\ a'] & = & \mathrm{wf}[a] \wedge \mathrm{wf}[a'] \wedge \mathrm{nozeros}(a')
\end{array}
$$

Definition 3 (inject) $\iota[\_] : \lambda \to \lambda'$ is a function that converts λ terms to λ' terms by changing all lets to 1-annotated lets:

$$
\begin{array}{lcl}
\iota[z] & = & z \\
\iota[n] & = & n \\
\iota[()] & = & () \\
\iota[\Omega] & = & \Omega \\
\iota[\pi_r\ e] & = & \pi_r\ \iota[e] \\
\iota[(e, e')] & = & (\iota[e], \iota[e']) \\
\iota[\lambda x{:}T.e] & = & \lambda x{:}T.\iota[e] \\
\iota[e\ e'] & = & \iota[e]\iota[e'] \\
\iota[\mathbf{let}\ z = e\ \mathbf{in}\ e'] & = & \mathbf{let}_1\ z = \iota[e]\ \mathbf{in}\ \iota[e']
\end{array}
$$

Definition 4 (erase) $\mathrm{erase}(\_) : \lambda' \to \lambda$ is a function that converts λ' terms to λ terms by erasing all annotations from lets:

$$
\begin{array}{lcl}
\epsilon[z] & = & z \\
\epsilon[n] & = & n \\
\epsilon[()] & = & () \\
\epsilon[\Omega] & = & \Omega \\
\epsilon[\pi_r\ a] & = & \pi_r\ \epsilon[a] \\
\epsilon[(a, a')] & = & (\epsilon[a], \epsilon[a']) \\
\epsilon[\lambda x{:}T.a] & = & \lambda x{:}T.\epsilon[a] \\
\epsilon[a\ a'] & = & \epsilon[a]\,\epsilon[a'] \\
\epsilon[\mathbf{let}_0\ z = a\ \mathbf{in}\ a'] & = & \mathbf{let}\ z = \epsilon[a]\ \mathbf{in}\ \epsilon[a'] \\
\epsilon[\mathbf{let}_1\ z = a\ \mathbf{in}\ a'] & = & \mathbf{let}\ z = \epsilon[a]\ \mathbf{in}\ \epsilon[a']
\end{array}
$$

Definition 5 (weak bisimulation) Given two transition systems $X \subseteq S \times S$ and $Y \subseteq S \times S$, we say that a relation $R \subseteq S \times S$ relating states of X to states of Y is a *weak simulation from X to Y* if and only if for every $e_x R e_y$ the following holds:

$$e_x \longrightarrow_X e'_x \implies \exists e'_y.\ e_y \longrightarrow^*_Y e'_y \ \wedge\ e'_x R e'_y$$

If R is a weak simulation from X to Y and a weak simulation from Y to X, then R is called a *weak bisimulation* between X and Y.

Here we introduce a function that expresses the correspondence between well-formed $\lambda_{r'}$ terms that were 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 alpha conversion.

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

Figure 18: instantiate-substitute correspondence

Definition 6 (extension of wf[−] to contexts) We extend wf[−] to act on A_1 contexts by including wf[_] = t and otherwise remaining unchanged from its action on expressions. On reduction contexts we define it as follows:

$$\begin{array}{lcl}
\mathrm{wf}[_] & = & \mathrm{t} \\
\mathrm{wf}[_.E_3] & = & \mathrm{wf}[E_3] \\
\mathrm{wf}[A_1.E_3] & = & \mathrm{wf}[A_1]\ \wedge\ \mathrm{wf}[E_3] \\
\mathrm{wf}[\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3] & = & \mathrm{wf}[\mathbf{let}_0\ z = u\ \mathbf{in}\ _]\ \wedge\ \mathrm{wf}[E_3]
\end{array}$$

Definition 7 (binding context)

$\mathcal{E}_c[E_3]^\Phi$ builds an environment corresponding to the binding context of the $\lambda_{r'}$ reduction context

E_3 using the environment Φ.

$$\begin{array}{lcl}
\mathcal{E}_c[_]^\Phi & = & \varnothing \\
\mathcal{E}_c[_.E_3]^\Phi & = & \mathcal{E}_c[E_3]^\Phi \\
\mathcal{E}_c[A_1.E_3]^\Phi & = & \mathcal{E}_c[E_3]^\Phi \\
\mathcal{E}_c[\mathbf{let}_0\ z = u\ \mathbf{in}\ _.E_3]^\Phi & = & z \mapsto [\![\ u\ ]\!]^\Phi, \mathcal{E}_c[E_3]^{\Phi, z \mapsto [\![\ u\ ]\!]^\Phi}
\end{array}$$

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.

Lemma 24 (well-formed properties) *wf*$[E_3.a] \iff$ *wf*$[E_3] \wedge$ *wf*$[a]$

Proof ($\Longrightarrow$) Assume $\mathrm{wf}[E_3.a]$ and note that $\mathrm{wf}[-]$ acts on contexts in the same way it acts on expressions, thus $\mathrm{wf}[E_3]$. Furthermore having a surrounding context can only impose stricter conditions upon a, thus $\mathrm{wf}[a]$.

($\Leftarrow$) Assume $\mathrm{wf}[E_3] \wedge \mathrm{wf}[a]$ and note that $\mathrm{wf}[-]$ can only fail if the nozeros or value checks fail. No holes in E_3 coincide with these checks, thus $\mathrm{wf}[E_3.a]$.

□

Lemma 25 (reduction preserves well-formedness) *wf*$[a] \wedge a \longrightarrow_{r'} a' \Longrightarrow$ *wf*$[a']$

Proof Prove this by showing that the transition system for $\lambda_{r'}$ is closed under $\mathrm{wf}[a] \Longrightarrow \mathrm{wf}[a']$ by rule induction on $a \longrightarrow_{r'} a'$.

case (proj) **:**

Assume $\mathrm{wf}[\pi_r\ (E_2.(u_1, u_2))]$, then by well-formed properties (Lemma 24) $\mathrm{wf}[E_2] \wedge \mathrm{wf}[(u_1, u_2)]$. By $\mathrm{wf}[-]$ definition we have $\mathrm{wf}[u_1] \wedge \mathrm{wf}[u_2]$. Thus by well-formed properties (Lemma 24) $\mathrm{wf}[E_2.u_r]$.

case (app) **:**

Assume $\mathrm{wf}[(E_2.\lambda x{:}T.a)\ u]$ then by definition: $\mathrm{wf}[E_2.\lambda x{:}T.a]$ and $\mathrm{wf}[u]$. We can deduce using well-formed properties (Lemma 24) and the definition of $\mathrm{wf}[-]$ that $\mathrm{wf}[E_2]$ and $\mathrm{wf}[a]$. It follows that $\mathrm{wf}[\mathbf{let}_0\ x = u\ \mathbf{in}\ a]$, then by well-formed properties (Lemma 24) $\mathrm{wf}[E_2.\mathbf{let}_0\ z = u\ \mathbf{in}\ a]$ as required.

case (omega) **:**

Immediate.

case (inst) **:**

Assuming $\mathrm{wf}[\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.z]$ we have by definition that $\mathrm{wf}[u]$ and $\mathrm{wf}[E_3.z]$, then by well-formed properties (Lemma 24) (used twice) we have $\mathrm{wf}[E_3.u]$ we then have by definition that $\mathrm{wf}[\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.u]$.

case (zero) **:**

Assuming $\mathrm{wf}[\mathbf{let}_1\ z = u\ \mathbf{in}\ a]$ we have by definition $\mathrm{wf}[u]$ and $\mathrm{wf}[a]$, thus $\mathrm{wf}[\mathbf{let}_0\ z = u\ \mathbf{in}\ a]$.

case (cong) **:**

Assume $\mathrm{wf}[a] \Longrightarrow \mathrm{wf}[a']$ and $\mathrm{wf}[E_3.a]$ then by well-formed properties (Lemma 24) $\mathrm{wf}[a]$ and thus using our inductive assumption $\mathrm{wf}[a']$. By well-formed properties (Lemma 24) $\mathrm{wf}[E_3.a']$.

□

B.1.1 Properties of substitute-instantiate correspondence

Lemma 26 **($[\![-]\!]^-$ environment properties)**

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

(ii) If $x \notin fv(a)$ *then* $[\![a]\!]^{\Phi,x\mapsto v} = [\![a]\!]^{\Phi}$

Proof First prove (i) by induction on a. Cases () and n are trivial.

case z **:**

Assume $\mathrm{fv}(z) \subseteq \mathrm{dom}(\Phi) \ \wedge\ \mathrm{fv}(v) \subseteq \mathrm{dom}(\Phi) \ \wedge\ \mathrm{wf}[z]$. We know by the definition of Φ that $\Phi(z) = z$ or $\Phi(z) = v'$ for some c-value v'. In the former case we have to consider if $z = x$ holds, if it does then

$$\{v/x\}[\![z]\!]^{\Phi,x\mapsto x} = \{x/v\}[\Phi, x \mapsto x](z) = \{v/x\}x = v = [\![z]\!]^{\Phi,x\mapsto v}$$

if not then

$$\{v/x\}[\![z]\!]^{\Phi,x\mapsto x} = \{x/v\}[\Phi, x \mapsto x](z) = \{v/x\}z = z = [\![z]\!]^{\Phi,x\mapsto v}$$

In the latter case

$$\{v/x\}[\![z]\!]^{\Phi,x\mapsto x} = \{x/v\}[\Phi, x \mapsto x](z) = \{v/x\}v'$$

holds and we are left to show that $\{v/x\}v' = v' = [\![z]\!]^{x\mapsto v,\Phi}$. The second equality is true by assumption. To show the first equality is suffices to prove $x \notin \mathrm{fv}(v')$. As $\Phi, x \mapsto x$ is an environment $x \notin \mathrm{dom}(\Phi)$ therefore given a z such that $\Phi(z) = v$ then $x \notin \mathrm{fv}(v)$ by the definition of environment.

case $\lambda z{:}T.a$ **:**

Assume $\mathrm{fv}(\lambda x{:}T.a) \subseteq \mathrm{dom}(\Phi)$ and $\mathrm{wf}[\lambda x{:}T.a]$. First note that by alpha conversion $z \neq x$ can be ensured. Then $\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi, x \mapsto x)$ and $\mathrm{wf}[a]$, so by induction

$$\{v/x\}[\![a]\!]^{\Phi,x\mapsto x} = [\![a]\!]^{\Phi,x\mapsto v}$$

From which the result follows by lambda abstracting on z.

The rest of the cases follow a similar pattern.

Part (ii) is clear from the definition of $[\![-]\!]^-$. □

Lemma 27 $fv(E_3.a) \subseteq dom(\Phi) \iff fv(a) \subseteq (dom(\Phi) \cup \mathcal{E}_c[E_3]^{\Phi})$

Proof ($\Longrightarrow$) Notice that $\mathrm{fv}(a)\backslash\mathrm{fv}(E_3.a)$ can be at most $\mathrm{hb}(E_3)$. The result is assured as $\mathrm{hb}(E_3) = \mathrm{dom}(\mathcal{E}_c[E_3]^{\Phi})$.

($\Leftarrow$) It suffices to observe that the hole binders of E_3 cannot be free in $E_3.a$ and that $\mathrm{hb}(E_3) = \mathrm{dom}(\mathcal{E}_c[E_3]^{\Phi})$.

□

Lemma 28 **($[\![-]\!]^-$ value preservation)**

$$fv(u) \subseteq dom(\Phi) \ \wedge\ wf[u] \implies [\![u]\!]^{\Phi}\ \mathsf{cval}$$

Proof Prove by induction on u. $[\![-]\!]^{\Phi}$ clearly preserves n and $()$, so these cases are trivial. In the pair case it acts inductively, and in the function case we transform functions in $\lambda_{r'}$ into functions in λ_c and functions are values. This leaves the let case:

case let$_0$ $= u_1$ in u_2 :
Assume fv($\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2$) $\subseteq$ dom(Φ) and wf[$\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2$]. $[\![\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2]\!]^{\Phi} = [\![u_2]\!]^{\Phi'}$ where $\Phi' = \Phi, z \mapsto [\![u_1]\!]^{\Phi}$. By Lemma 27 fv($u_2$) $\subseteq$ dom(Φ') and by definition of wf[$-$], wf[u_2]. By induction $[\![u_2]\!]^{\Phi'}$ cval as required.

□

Definition 8 ($[\![-]\!]$ on contexts) We extend $[\![-]\!]^{-}$ to act on A_1 contexts by adding the clause $[\![\_]\!]^{\Phi} = \_$. On reduction contexts we define the action as:

$$\begin{aligned}
[\![\mathbf{let}_0\ z = a\ \mathbf{in}\ _.E_3]\!]^{\Phi} &= [\![E_3]\!]^{\Phi, z\mapsto [\![a]\!]^{\Phi}} \\
[\![A_1.E_3]\!]^{\Phi} &= [\![A_1]\!]^{\Phi}.[\![E_3]\!]^{\Phi} \\
[\![_.E_3]\!]^{\Phi} &= _.[\![E_3]\!]^{\Phi}
\end{aligned}$$

Lemma 29 ($[\![-]\!]$ distribution over contexts) *For all E_3, Φ and a, if fv($E_3.a$) $\subseteq$ dom(Φ) and wf[$E_3.a$] then $[\![E_3.a]\!]^{\Phi} = [\![E_3]\!]^{\Phi}.[\![a]\!]^{\Phi, \mathcal{E}_c[E_3]^{\Phi}}$*

Proof We prove by induction on E_3.

case $A_1.E_3'$:

$$\begin{aligned}
[\![A_1.E_3'.a]\!]^{\Phi} &= [\![A_1]\!]^{\Phi}.\ [\![E_3'.a]\!]^{\Phi} \\
&= [\![A_1]\!]^{\Phi}.\ [\![E_3']\!]^{\Phi}.\ [\![a]\!]^{\Phi, \mathcal{E}_c[E_3']^{\Phi}} \quad (*) \\
&= [\![A_1.E_3']\!]^{\Phi}.\ [\![a]\!]^{\Phi, \mathcal{E}_c[A_1.E_3']^{\Phi}}
\end{aligned}$$

By well-formed properties (Lemma 24) we have wf[$E_3'.a$], and by assumption fv($E_3'.a$) $\subseteq$ dom(Φ), thus by induction (*) holds.

case let$_0$ $z = u$ in $\_.E_3'$:

$$\begin{aligned}
[\![\mathbf{let}_0\ z = u\ \mathbf{in}\ _.E_3'.a]\!]^{\Phi} &= [\![E_3'.a]\!]^{\Phi, z\mapsto [\![u]\!]^{\Phi}} \\
&= [\![E_3']\!]^{\Phi, z\mapsto [\![u]\!]^{\Phi}}.\ [\![a]\!]^{\Phi'} \quad (*) \\
&\qquad \text{where } \Phi' = \Phi, z \mapsto [\![u]\!]^{\Phi}, \mathcal{E}_c[E_3']^{\Phi, z\mapsto [\![u]\!]^{\Phi}} \\
&= [\![\mathbf{let}_0\ z = u\ \mathbf{in}\ _.E_3']\!]^{\Phi}.\ [\![a]\!]^{\Phi, \mathcal{E}_c[\mathbf{let}_0\ z=u\ \mathbf{in}\ _.E_3']^{\Phi}} \quad (**)
\end{aligned}$$

By definition of wf[$-$] we have wf[$E_3'.a$], and by assumption fv($E_3'.a$) $\subseteq$ dom(Φ'), thus by induction (*) holds. By definition of $[\![-]\!]^{-}$ and $\mathcal{E}_c[-]^{-}$, (**) is equivalent to (*).

□

Lemma 30 ($[\![-]\!]$ preserves contexts) *If fv(E_3) $\subseteq$ dom(Φ) and wf[E_3] then there exists a λ_c reduction context E such that $[\![E_3]\!]^{\Phi} = E$.*

Proof We proceed by induction on the structure of E_3:

case $\_$:
$[\![\_]\!]^{\Phi} = \_$ which is a valid λ_c reduction context.

case $A_1.E_3'$ **:**

Assume (1) $\text{fv}(A_1.E_3') \subseteq \text{dom}(\Phi)$ and (2) $\text{wf}[A_1.E_3']$. From $\llbracket - \rrbracket$ on contexts (definition 9) $\llbracket A_1.E_3' \rrbracket^\Phi = \llbracket A_1 \rrbracket^\Phi.\llbracket E_3 \rrbracket^\Phi$. Clearly $\text{fv}(E_3') \subseteq \text{dom}(\Phi)$ and by well-formed properties (Lemma 24) $\text{wf}[E_3']$. From these derived facts and induction, there exists an E' such that $E = \llbracket E_3' \rrbracket^\Phi$. We are left to show that $\llbracket A_1 \rrbracket^\Phi$ is a valid λ_c reduction context for every A_1:

case $(\_, a)$ **:**

Follows directly from definition

case $(u, \_)$ **:**

$\llbracket (u, \_) \rrbracket^\Phi = (\llbracket u \rrbracket^\Phi, \_)$ which is a λ_c context only if $\llbracket u \rrbracket^\Phi$ cval. From 1 we know that $\text{fv}(u) \subseteq \text{dom}(\Phi)$ as $\text{fv}(u) \subseteq \text{fv}((u,\_).E_3')$. From 2 we conclude $\text{wf}[u]$. By these last two facts and $\llbracket - \rrbracket^-$ value preservation (Lemma 28) $\llbracket u \rrbracket^\Phi$ cval as required.

The rest of the A_1 cases are similar to one of the above two.

case $\textbf{let}_0\ z = u\ \textbf{in}\ \_.E_3'$ **:**

Assume $\text{fv}(\textbf{let}_0\ z = u\ \textbf{in}\ \_.E_3') \subseteq \Phi$ and $\text{wf}[\textbf{let}_0\ z = u\ \textbf{in}\ \_.E_3']$. From $\llbracket - \rrbracket$ on contexts (definition 9) $\llbracket \textbf{let}_0\ z = u\ \textbf{in}\ E_3' \rrbracket^\Phi = \llbracket E_3' \rrbracket^{\Phi, z \mapsto \llbracket u \rrbracket^\Phi}$. It is clear that $\text{fv}(E_3') \subseteq \text{dom}(z \mapsto \llbracket u \rrbracket \Phi)\Phi$, and by well-formed properties (Lemma 24) $\text{wf}[E_3]$, thus by induction there exists an E such that $\llbracket E_3 \rrbracket^{\Phi, z \mapsto \llbracket u \rrbracket^\Phi} = E$.

□

Definition 9 **(** $\epsilon[-]$ **on contexts)** We extend $\epsilon[-]$ to act on A_1 contexts by adding the clause $\epsilon[\_] = \_$ On reduction contexts we define the action as:

$$\begin{array}{rcl} \epsilon[\textbf{let}_0\ z = a\ \textbf{in}\ \_.E_3] & = & \textbf{let}\ z = \epsilon[a]\ \textbf{in}\ \epsilon[E_3] \\ \epsilon[A_1.E_3] & = & \epsilon[A_1].\epsilon[E_3] \\ \epsilon[\_.E_3] & = & \_.\epsilon[E_3] \end{array}$$

Notation we write $a \xrightarrow{x} a'$ to indicated that rule x was used to reduce a to a'.

Definition 10 **(INF)** A term a is in *instantiation normal form* (INF) if and only if there does not exist an a' such that $a \xrightarrow{\text{inst}} a'$. We write a $\mathsf{inf_r}$ when a is in INF.

Definition 11 **(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.

Lemma 31 **(** $\mathsf{inf}_\mathsf{r}^\circ$ **preserved by** E_3 **stripping)** *For any evaluation context* E_3, $E_3.a\ \mathit{inf}_r^\circ \implies a\ \mathit{inf}_r^\circ$

Proof Proof of the contrapositive follows simply. □

Lemma 32 **(** $\llbracket - \rrbracket^-$ **invariant under insts)** $\mathit{wf}[a] \wedge \mathit{fv}(a) \subseteq \mathit{dom}(\Phi) \wedge a \xrightarrow{inst}{}^*_{r'} a' \implies \llbracket a \rrbracket^\Phi = \llbracket a' \rrbracket^\Phi$

Proof We first prove the single reduction case by induction on $a \xrightarrow{\text{insts}}_{r'} a'$. Every case is trivial except (inst) and (cong):

case (inst) **:**

Assume $\text{wf}[\textbf{let}_0\ z = u\ \textbf{in}\ E_3.z]$ and $\text{fv}(\textbf{let}_0\ z = u\ \textbf{in}\ E_3.z) \subseteq \text{dom}(\Phi)$. We are required to prove that applying $\llbracket - \rrbracket^\Phi$ to the left and right hand side of this rule results in the same term. First take the LHS:

$$\begin{array}{rcll} \llbracket \textbf{let}_0\ z = u\ \textbf{in}\ E_3.z \rrbracket^\Phi & = & \llbracket E_3.z \rrbracket^{\Phi'} & \\ & & \text{where } \Phi' = \Phi, z \mapsto \llbracket u \rrbracket^\Phi & \\ & = & \Phi', \llbracket z \rrbracket^{\mathcal{E}_c[E_3]^{\Phi'}} & (\dagger) \\ & = & \llbracket u \rrbracket^\Phi & (*) \end{array}$$

(†) follows from $[\![-]\!]^{-}$ distribution over contexts (Lemma 29) . (*) follows as $z \notin \mathrm{hb}(E_3)$ by the side condition of rule. Now take the RHS:

$$[\![\mathbf{let}_0\ z = u \ \mathbf{in}\ E_3.u]\!]^{\Phi} \quad = \quad [\![u]\!]^{\Phi', \mathcal{E}_c[E_3]^{\Phi'}} \qquad (**)$$

We are left to show that (*) and (**) are equal.

By side condition of the (inst) reduction rule $\mathrm{fv}(u) \notin \mathrm{hb}(E_3)$ and by alpha conversion $z \notin \mathrm{fv}(u)$. It follows that $\mathrm{fv}(u) \notin \mathrm{dom}(\Phi', \mathcal{E}_c[E_3]^{\Phi'}) \cup z$, thus by induction on the number of bindings in $\Phi', \mathcal{E}_c[E_3]^{\Phi'}$ we can show, using $[\![-]\!]^{-}$ environment properties (ii) (Lemma 26) , that $[\![u]\!]^{\Phi} = [\![u]\!]^{\Phi', \mathcal{E}_c[E_2]^{\Phi'}}$, as required.

case (cong) **:**

Assume $\mathrm{wf}[E_3.a]$ and $\mathrm{fv}(E_3.a) \subseteq \mathrm{dom}(\Phi)$. By well-formed properties (Lemma 24) $\mathrm{wf}[a]$. Let $\Phi' = \Phi, \mathcal{E}_c[E_3]^{\Phi}$, then $\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi')$. By induction $[\![a]\!]^{\Phi'} = [\![a']\!]^{\Phi'}$ (*). Now $[\![E_3.a]\!]^{\Phi} = [\![a]\!]^{\Phi'}$ and $[\![E_3.a']\!]^{\Phi} = [\![a']\!]^{\Phi'}$, thus by (*) we are done.

The multiple step case follows by induction on the number of reductions. □

Definition 12 ($\mathrm{instvar}[-]$) $\mathrm{instvar}[a]$ denotes the number of potential instantiations that a can do.

$$\begin{array}{rcl}
\mathrm{instvar}[z] & = & 1 \\
\mathrm{instvar}[n] & = & 0 \\
\mathrm{instvar}[()] & = & 0 \\
\mathrm{instvar}[\Omega] & = & 0 \\
\mathrm{instvar}[\pi_r\ a] & = & \mathrm{instvar}[a] \\
\mathrm{instvar}[(a\ a')] & = & \mathrm{instvar}[a] + \mathrm{instvar}[a'] \\
\mathrm{instvar}[\lambda x{:}T.a] & = & 0 \\
\mathrm{instvar}[aa'] & = & \mathrm{instvar}[a] + \mathrm{instvar}[a'] \\
\mathrm{instvar}[\mathbf{let}_m\ z = a \ \mathbf{in}\ a'] & = & \mathrm{instvar}[a] + \mathrm{instvar}[a']
\end{array}$$

Lemma 33 ($\mathrm{instvar}[-]$ **properties)** *For all $\lambda_{r'}$ terms a and a'*

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

Proof First prove 1: For $\mathrm{instvar}[u]$ to be non-zero, there must be at least one occurrence of a variable that is not under a lambda binding. By the definition of the forms of values, this cannot be the case.

Now prove 2: Assume $a \xrightarrow{\mathrm{inst}}_{r'} a'$ and prove $\mathrm{instvar}[a'] = \mathrm{instvar}[a] - 1$. By the assumption the following must hold:

(3) $\exists E_3, E_3', z, u.\ a = E_3.\mathbf{let}_0\ z = u \ \mathbf{in}\ E_3'.z$

we are left to prove

$$\mathrm{instvar}[E_3.\mathbf{let}_0\ z = u \ \mathbf{in}\ E_3'.u] = \mathrm{instvar}[E_3.\mathbf{let}_0\ z = u \ \mathbf{in}\ E_3'.z] - 1$$

which is true if and only if $\mathrm{instvar}[u] = \mathrm{instvar}[z] - 1$, which holds if and only if $\mathrm{instvar}[u] = 0$, which is assured by our first observation.

□

Lemma 34 (INF reachability) *For all closed a, if wf$[a]$ then there exists a' such that $a \xrightarrow{insts}{}^{*}_{r'} a' \wedge a'$ $\mathsf{inf_r}$*

Proof Assume a closed and wf$[a]$. If a does not match the LHS of an inst or instrec rule then we are done, so suppose that it does. By instvar$[-]$ properties (Lemma 33) there can only be finitely many inst or instrec reductions, say n. Thus after n insts reductions we arrive at a term a', for which it must hold that a' does not match the LHS of inst or instrec and thus a' $\mathsf{inf_r}$ as required. □

Lemma 35 ($[\![-]\!]^\Phi$ source-value property) *For all $\lambda_{r'}$ expressions a, the following holds:*

$$wf[a] \;\wedge\; a\ \mathsf{inf}_r^\circ \;\wedge\; fv(a) \subseteq dom(\Phi) \;\wedge\; [\![\, a \,]\!]^\Phi\ \mathsf{cval} \implies a\ \mathsf{r'val}$$

Proof We prove by induction on a. In the identifier case, the term is not in INF. The n and $()$ cases are immediate. The (a_1, a_2) case follows by induction (and that the subterms are well-formed). The $\pi_r\ a$ and $a_1\ a_2$ cases are immediate as the action of $[\![-]\!]^\Phi$ on them does not produce a value. The function case is also immediate as $[\![-]\!]^\Phi$ produces a function which is a value. In the $\mathbf{let}_1$ case, applying $[\![-]\!]^\Phi$ does not produce a value. This leaves the let case:

case $\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2$:

Assume the term is well-formed, then the subterms are well-formed and a_1 $\mathsf{r'val}$. Assume the term is in open INF, then a_2 inf_r°. Assume that the free variables of the term are in dom(Φ), then fv$(a_2) \subseteq \Phi, z \mapsto [\![\, a_1 \,]\!]^\Phi$. We have to prove:

$$[\![\, \mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2 \,]\!]^\Phi = [\![\, a_2 \,]\!]^{\Phi, z \mapsto [\![\, a_1 \,]\!]^\Phi}$$

is an r-value, which follows by induction on a_2.

□

Lemma 36 ($[\![-]\!]$ outer value preservation) *For all $\lambda_{r'}$ values u:*

(a) If wf$[u]$, fv$(a) \subseteq$ dom(Φ) and $[\![\, u \,]\!]^\Phi = \lambda x{:}T.e$ then there exists E_2, a, x such that $u = E_2.\lambda x{:}T.a$

(b) $[\![\, u \,]\!]^\Phi = (v_1, v_2) \implies \exists E_2, u_1, u_2.\ u = E_2.(u_1, u_2)$

Proof We prove (a) by induction on u. The cases of $n, (), (u_1, u_2)$ are trivially true as $[\![-]\!]^-$ on these terms can not result in a term of the form $\lambda x{:}T.e$. The case $\lambda x{:}T.a$ results in a function when $[\![-]\!]^-$ is applied, but it is already of the right form if one chooses $E_2 = \_$. This leaves the let case:

case $\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2$:

Assume wf$[\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2]$; fv$(\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2) \subseteq$ dom(Φ) and $[\![\, \mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2 \,]\!]^\Phi = \lambda x{:}T.e$. By definition of well-formedness wf$[u_1] \wedge$ wf$[u_2]$. It is easy to see that fv$(u_1) \subseteq$ dom(Φ) and fv$(u_2) \subseteq$ dom$(\Phi, z \mapsto [\![\, u_1 \,]\!]^\Phi)$. From the last assumption $[\![\, \mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2 \,]\!]^\Phi = [\![\, u_2 \,]\!]^{\Phi, z \mapsto [\![\, u_1 \,]\!]^\Phi} = \lambda x{:}T.e$, thus by induction there exists E_2', a', x' such that $u_2 = E_2'.\lambda x'{:}T.a'$. The result follows by choosing $E_2 = (\mathbf{let}_0\ z = u_1\ \mathbf{in}\ \_.E_2')$; $a = a'$ and $x = x'$.

(b) is proved by a similar induction on u. □

B.1.2 Erase properties

Definition 13 (ZNF) A $\lambda_{r'}$ expression is in *zero normal form*, denoted by a $\mathsf{znf_r}$ if and only if there does not exist an a' such that $a \xrightarrow{\text{zero}}_{r'} a'$.

Definition 14 (open ZNF) We say that a possibly open $\lambda_{r'}$ expression is in *open zero normal form* and write a $\mathsf{znf_r^\circ}$ 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 ($\mathsf{znf_r^\circ}$ preserved by E_3 stripping) $E_3.a$ *znf*$_r^\circ$ $\implies$ a *znf*$_r^\circ$

Proof Proof is easily obtained by proving the contrapositive. □

Lemma 38 ($\epsilon[-]$ invariant under zeros) $wf[a] \wedge a \xrightarrow{zero}{}^*_{r'} a' \implies \epsilon[a] = \epsilon[a']$

Proof Observe that both sides of the (zero) rule erase to the same term.

□

Lemma 39 (RZNF reachability) *For all closed a, if $wf[a]$ then there exists a' such that $a \xrightarrow{zero}{}^*_{r'} a' \wedge a'$ znf$_r$*

Proof To see this we show that all contiguous sequences of (zero)-reductions are finite. Define a metric ones: $\lambda' \to \mathbb{N}$ that counts the number of 1-annotated-lets 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.

□

Lemma 40 ($\epsilon[-]$ value preservation)

$$wf[u] \implies \epsilon[u]\ \mathit{rval}$$

Proof Obvious from definition. □

Lemma 41 ($\epsilon[-]$ distributes over contexts) $\epsilon[E_3.a] = \epsilon[E_3].\epsilon[a]$

Proof Straight forward induction on E_3. □

Lemma 42 ($\epsilon[-]$ preserves contexts) *If $wf[E_3]$ then there exists a λ_r reduction context E_3' such that $\epsilon[E_3] = E_3'$.*

Proof By induction on E_3:

case _ :

trivial

case $A_1.E_3$:

Assume that wf$[A_1.E_3]$. By well-formed properties (Lemma 24) wf$[E_3]$. By induction there exists a λ_r context E_3' such that $\epsilon[E_3] = E_3'$. Now $\epsilon[A_1.E_3] = \epsilon[A_1].\epsilon[E_3] = \epsilon[A_1].E_3'$ and furthermore, by $\epsilon[-]$ value preservation (Lemma 40) it is easy to verify that for each A_1, $\epsilon[A_1]$ is a valid λ_r atomic context.

case $\mathbf{let}_0\ z = u\ \mathbf{in}\ \_.E_3$ **:**

Similar to the previous case.

□

Lemma 43 ($\epsilon[-]$ **source-value property)** $\mathit{wf}[a] \wedge a\ \mathsf{znf}_r^\circ \wedge \epsilon[a]\ \mathsf{rval} \implies a\ \mathsf{r'val}$

Proof Straightforward induction on a. □

Lemma 44 ($\epsilon[-]$ **outer value preservation)** *For all* $\lambda_{r'}$ *values* u*:*

(a) If $\mathit{wf}[u]$ *and* $\epsilon[u] = E_2.\lambda x{:}T.e$ *then there exists* $\hat{E}_2, a, z$ *such that one of the following holds:*

(i) $u = \hat{E}_2.\lambda x{:}T.a$

(b) $\epsilon[u] = E_2.(v_1, v_2) \implies \exists \hat{E}_2, u_1, u_2.\ u = \hat{E}_2.(u_1, u_2)$

Proof Follows by inspection of the definition. □

Lemma 45 ($\epsilon[-]$ **source context)** *If* $\epsilon[a] = E_3.e$ *and* $a\ \mathsf{znf}_r^\circ$ *then there exists an* $\hat{E}_3$ *and* $\hat{a}$ *such that* $a = \hat{E}_3.\hat{a}$ *and* $\epsilon[\hat{E}_3] = E_3$*.*

Proof Proceed by induction on E_3, we show only a sample of the cases as the rest are similar:

case $(v, \_).E_3$ **:**

Assume $\epsilon[a] = (v, \_).E_3.e$ and $a\ \mathsf{znf}_r^\circ$. The only possible form for a is (a_1, a_2) for some a_1 and a_2. Thus $\epsilon[a_1] = v$ and $\epsilon[a_2] = E_3.e$. As a is in open ZNF, a_1 must also be, thus by $\epsilon[-]$ source-value property (Lemma 43) $a_1\ \mathsf{r'val}$. By induction on E_3 there exists $\hat{E}_3$ and $\hat{a}$ such that $a_2 = \hat{E}_3.\hat{a} \wedge \epsilon[\hat{E}_3] = E_3$. It follows that $(a_1, a_2) = (a_1, \_).\hat{E}_3.\hat{a} \wedge \epsilon[(a_1, \_).\hat{E}_3.\hat{a}] = (v, \_).E_3$.

case $\mathbf{let}\ z = u\ \mathbf{in}\ E_3$ **:**

Assume $\epsilon[a] = \mathbf{let}\ z = u\ \mathbf{in}\ E_3.e$ and $a\ \mathsf{znf}_r^\circ$. By inspection of the definition of $\epsilon[-]$ a either has the form $\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2$ or $\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2$ for some a_1 or a_2. The latter cannot be the case, as assume that it is, then by znf_r° preserved by E_3 stripping (Lemma 37) $a_1\ \mathsf{znf}_r^\circ$, but $\epsilon[a_1] = u$ so by $\epsilon[-]$ source-value property (Lemma 43) $a_1\ \mathsf{r'val}$ and so $\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2$ in not in open ZNF, a contradiction. We continue considering $a = \mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2$. We have $\epsilon[a_1] = u$, $\epsilon[a_2] = E_3.e$ and as wf$[a]$, $a_1\ \mathsf{r'val}$. By induction on E_3 there exists an $\hat{E}_3$ and $\hat{a}$ such that $a_2 = \hat{E}_3.\hat{a} \wedge \epsilon[\hat{E}_3] = E_3$. It follows that $a = \mathbf{let}\ z = a_1\ \mathbf{in}\ \hat{E}_3.\hat{a}$ and $\epsilon[\mathbf{let}\ z = a_1\ \mathbf{in}\ \hat{E}_3] = \mathbf{let}\ z = u\ \mathbf{in}\ E_3$ as required.

□

Lemma 46 (inst match property)

$$\mathit{wf}[a] \wedge a \xrightarrow{inst}_{r'} a' \implies \exists e'.\ \epsilon[a] \xrightarrow{inst}_{r} e' \wedge e' = \epsilon[a']$$

Proof We prove by induction on the structure of $a \xrightarrow{\text{insts}}_{r'} a'$:

case (inst) **:**

$$\begin{array}{ll} & \epsilon[\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.z] \\ = & \mathbf{let}\ z = \epsilon[u]\ \mathbf{in}\ \epsilon[E_3].\epsilon[z] \\ \longrightarrow_r & \mathbf{let}\ z = \epsilon[u]\ \mathbf{in}\ \epsilon[E_3].\epsilon[u] \\ = & \epsilon[\mathbf{let}\ z = u\ \mathbf{in}\ E_3.u] \end{array}$$

Where the penultimate step is valid by $\epsilon[-]$ preserves contexts (Lemma 42) .

case (cong) **:**

Assume wf$[E_3.a]$; $E_3.a \longrightarrow_{r'} E_3.a'$ and $a \xrightarrow{\text{insts}}_{r'} a'$. It follows from well-formed properties (Lemma 24) that wf$[a]$. By induction on a there exists an e' such that $\epsilon[a] \xrightarrow{\text{insts}}_{r'} e'$ and $e' = \epsilon[a']$. As $\epsilon[E_3]$ is a valid λ_r context by $\epsilon[-]$ preserves contexts (Lemma 42) $\epsilon[E_3].\epsilon[a] \longrightarrow_r \epsilon[E_3].e'$. To get the result it is sufficient to prove that $\epsilon[E_3].e' = \epsilon[E_3.a']$. It follows from $\epsilon[-]$ distributes over contexts (Lemma 41) that $\epsilon[E_3.a'] = \epsilon[E_3].\epsilon[a'] = \epsilon[E_3].e'$ as required.

□

Lemma 47 (inst match sequence)

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

Proof By induction on the length of the transition sequence (n):

case $n = 0$ **:**

Immediate.

case $n = k$ **:**

Assume (4) wf$[a]$ $\wedge$ $a \xrightarrow{\text{inst}}{}^{k+1}_{r} a'$ and prove (5) $\exists e'.\ \epsilon[a] \xrightarrow{\text{inst}}{}^{k+1}_{r'} e' \wedge e' = \epsilon[a']$. By 4 $\exists a''.\ a \xrightarrow{\text{inst}}{}^{k}_{r} a'' \xrightarrow{\text{inst}}_{r} a'$ thus by IH (6) $\exists e''.\ \epsilon[a] \xrightarrow{\text{inst}}{}^{k}_{r} e'' \wedge e'' = \epsilon[a'']$. Recall that well-formedness is preserved by reduction so wf$[a'']$. By the above results and inst match property (Lemma 46) we have (7) $\exists e'.\ \epsilon[a''] \xrightarrow{\text{insts}}_{r} e' \wedge e' = \epsilon[a']$, thus by 6 and 7: $\exists e'.\ \epsilon[a] \xrightarrow{\text{insts}}{}^{k+1}_{r'} e' \wedge e' = \epsilon[a']$ as required.

□

Lemma 48 (zero match property)

$$wf[a] \;\wedge\; fv(a) \subseteq dom(\Phi) \;\wedge\; a \xrightarrow{zero}_{r'} a' \implies \exists e'.\ [\![\, a \,]\!]^{\Phi} \xrightarrow{let}_{c} e' \;\wedge\; e' = [\![\, a' \,]\!]^{\Phi}$$

Proof We prove by induction on the structure of $a \xrightarrow{\text{zero}}_{r'} a'$:

case (zero) **:**

observe

$$\begin{array}{rcll} [\![\, \mathbf{let}_1\ z = u\ \mathbf{in}\ a \,]\!]^{\Phi} & = & \mathbf{let}\ z = [\![\, u \,]\!]^{\Phi}\ \mathbf{in}\ [\![\, a \,]\!]^{\Phi, z \mapsto z} & \\ & \longrightarrow_c & \{[\![\, u \,]\!]^{\Phi}/z\}[\![\, a \,]\!]^{\Phi, x \mapsto x} & \\ & = & [\![\, a \,]\!]^{\Phi, x \mapsto [\![\, u \,]\!]^{\Phi}} & (*) \\ & = & [\![\, \mathbf{let}_0\ z = u\ \mathbf{in}\ a \,]\!]^{\Phi} & \end{array}$$

where step (*) is allowed by $[\![-]\!]^{-}$ environment properties (i) (Lemma 26) .

case (cong) **:**

Assume wf$[E_3.a]$; fv$(E_3.a) \subseteq$ dom(Φ); $E_3.a \xrightarrow{\text{zero}}_{r'} E_3.a'$. Notice that $[\![\, E_3.a \,]\!]^{\Phi} = [\![\, E_3 \,]\!]^{\Phi}.[\![\, a \,]\!]^{\Phi, \mathcal{E}_c[E_3]^{\Phi}}$. We can derive fv$(a) \subseteq$ dom$(\Phi, \mathcal{E}_c[E_3]^{\Phi})$. By well-formed properties (Lemma 24) wf$[a]$. By induction there exists an e' such that $[\![\, a \,]\!]^{\Phi, \mathcal{E}_c[E_3]^{\Phi}} \longrightarrow_c e' \wedge e' = [\![\, a' \,]\!]^{\Phi, \mathcal{E}_c[E_3]^{\Phi}}$. By $[\![-]\!]$ preserves contexts (Lemma 30) there exists a λ_c context such that $[\![\, E_3 \,]\!]^{\Phi} = E$, thus $E.[\![\, a \,]\!]^{\Phi, \mathcal{E}_c[E_3]^{\Phi}} \longrightarrow_c E.e'$. It now remains to show that $E.e' = [\![\, a' \,]\!]^{\Phi, \mathcal{E}_c[E_3]^{\Phi}}$, and this is assured by $[\![-]\!]^{-}$ distribution over contexts (Lemma 29) .

□

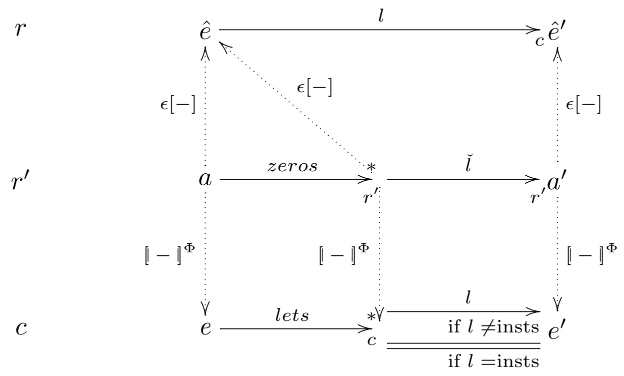

Figure 19: Operational reasoning of rc-simulation

Lemma 49 (zero match sequence)

$$wf[a] \ \wedge\ fv(a) \subseteq dom(\Phi) \ \wedge\ a \xrightarrow{zero}{}^{n}_{r'} a' \implies \exists e'.\ [\![\, a \,]\!]^{\Phi} \xrightarrow{let}{}^{n}_{c} e' \ \wedge\ e' = [\![\, a' \,]\!]^{\Phi}$$

Proof Proceed by induction on the length of transitions:

case $n = 0$ **:**
Immediate.

case $n = k+1$ **:**
Assume wf$[a] \wedge$ fv$(a) \subseteq$ dom$(\Phi) \wedge a \xrightarrow{\text{zero}}{}^{k+1}_{r'} a'$. By reduction we are assured that there exists an a'' such that $a \xrightarrow{\text{zero}}{}^{k}_{r'} a''$. By induction there exists an e' such that $[\![\, a \,]\!]^{\Phi} \xrightarrow{\text{let}}{}^{k}_{c} e' \wedge e' = [\![\, a'' \,]\!]^{\Phi}$ (*). As reduction can only remove variables from the set of free variables of a term we have fv$(a) \subseteq$ dom(Φ). By zero match property (Lemma 48) there exists an e'' such that $[\![\, a'' \,]\!]^{\Phi} \xrightarrow{\text{let}}_{c} e'' \wedge e'' = [\![\, a' \,]\!]^{\Phi}$ (**). By (*), (**) we have the result.

□

B.2 Bisimulation

The demonstration of a bisimulation between λ_c and λ_r will show that if an expression has a terminating reduction sequence under both systems, then the results will be related. In order to show observational equivalence we are, of course, left to show that termination in one system must lead to termination in the other. We first concentrate on showing that the relation defined in definition 16 is a weak bisimulation. To do this we prove it is a weak simulation from λ_r to λ_c by establishing the commutativity of the diagram in figure 19 and similarly the converse is established by demonstrating the commutativity of the diagram in figure 20.

Definition 15 (Candidate bisimulation)

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

Definition 16 (id$_\lambda$) id$_\lambda$ is the identity relation on lambda terms:

$$\text{id}_\lambda = \{(e, e) \mid e \ \textbf{in}\ \lambda \ \wedge\ e \ \mathsf{closed}\}$$

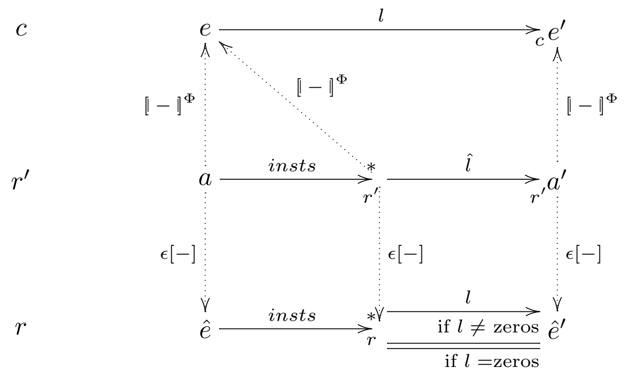

Figure 20: Operational reasoning of cr-simulation

Lemma 50 ($\mathbf{id}_\lambda \subseteq R$) *The candidate bisimulation R contains id_λ.*

Proof It suffices to prove $\epsilon[\iota[e]] = e$ and $[\![\, \iota[e] \,]\!]^{\varnothing} = e$. The first is clear from the definitions. The second can be proved by induction on e. ☐

B.2.1 c-r correspondence

Lemma 51 (c-r' correspondence)

$$a \textsf{ closed} \wedge wf[a] \wedge [\![\, a \,]\!]^{\varnothing} \longrightarrow_c e' \implies \exists a', a''.\ a \xrightarrow{insts}{}^{*}_{r'} a'' \longrightarrow_{r'} a' \wedge a'' \textsf{ inf}_{\textsf{r}} \wedge e' = [\![\, a' \,]\!]^{\varnothing}$$

Proof We generalise to open terms and claim that it is sufficient to prove:

$$\mathrm{wf}[a] \wedge \mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi) \wedge a \textsf{ inf}_{\textsf{r}}^{\circ} \wedge [\![\, a \,]\!]^{\Phi} \longrightarrow_c e' \implies \exists a'.\ a \longrightarrow_{r'} a' \wedge e' = [\![\, a' \,]\!]^{\Phi}$$

First let us show that this is sufficient: assume the above proposition and a $\textsf{closed}$ $\wedge$ wf$[a]$ $\wedge$ $[\![\, a \,]\!]^{\varnothing} \longrightarrow_c e'$ then we are required to prove that there exists an a' and a'' such that (8) $a \xrightarrow{\text{insts}}{}^{*}_{r'} a''$; (9) fv$(a) \subseteq$ dom(Φ); (10) $a'' \longrightarrow_{r'} a'$; (11) a'' $\textsf{inf}_{\textsf{r}}$ and (12) $e' = [\![\, a' \,]\!]^{\varnothing}$. By INF reachability lemma (Lemma 34) there exists an a'' to satisfy 8 and 11, thus taking $\Phi = \varnothing$ in the generalised claim by modus ponens we have that there exists an a' such that $a'' \longrightarrow_{r'} a' \wedge e' = [\![\, a' \,]\!]^{\varnothing}$. This satisfies the remaining proof obligations.

We prove the generalised claim by induction on the structure of a:

case z **:**
$\neg(z \textsf{ inf}_{\textsf{r}}^{\circ})$.

case $n; ()$ **:**
$[\![\, n \,]\!]^{\Phi} = n$ which does not reduce under λ_c. $[\![\, () \,]\!]^{\Phi} = ()$ which does not reduce under λ_c.

case (a_1, a_2) **:**
Assume wf$[(a_1, a_2)]$ $\wedge$ (a_1, a_2) $\textsf{inf}_{\textsf{r}}^{\circ}$ $\wedge$ $[\![\, (a_1, a_2) \,]\!]^{\Phi} \longrightarrow_c e'$ and prove that there exists an a' such that $(a_1, a_2) \longrightarrow_{r'} a' \wedge e' = [\![\, a' \,]\!]^{\Phi}$. We proceed by case split on the reductions of $[\![\, (a_1, a_2) \,]\!]^{\Phi}$.

case $[\![\, (a_1, a_2) \,]\!]^{\Phi} \longrightarrow_c (e_1', [\![\, a_2 \,]\!]^{\Phi})$ **:**

If follows that $[\![a_1]\!]^\Phi \longrightarrow_c e_1'$. By wf[$-$] definition wf[$a_1$]. We can reason that $\mathrm{fv}(a_1) \subseteq \mathrm{dom}(\Phi)$. By $\mathsf{inf}^\circ_\mathsf{r}$ preserved by E_3 stripping (Lemma 31) a_1 $\mathsf{inf}^\circ_\mathsf{r}$. By induction $a_1 \longrightarrow_{r'} a_1' \wedge [\![a_1]\!]^\Phi = e_1'$ (*). Thus $a_1\, a_2 \longrightarrow_{r'} a_1'\, a_2$ and we are left to show that the erasure of the RHS of this is equal to $(e_1', [\![a_2]\!]^\Phi)$: $[\![a_1'\, a_2]\!]^\Phi = ([\![a_1']\!]^\Phi, [\![a_2]\!]^\Phi) = (e_1', [\![a_2]\!]^\Phi)$ as required.

case $[\![(a_1, a_2)]\!]^\Phi \longrightarrow_c ([\![a_1]\!]^\Phi, e_2')$ **:**
Similar to last case.

case $\pi_r\, a$ **:**
Assume $\mathrm{wf}[\pi_r\, a] \wedge \pi_r\, a\ \mathsf{inf}^\circ_\mathsf{r} \wedge [\![\pi_r\, a]\!]^\Phi \longrightarrow_c e'$ and prove that there exists an a' such that $\pi_r\, a \longrightarrow_{r'} a' \wedge e' = [\![a']\!]^\Phi$. We proceed by case split on the reductions of $[\![\pi_r\, a]\!]^\Phi$.

case $[\![\pi_r\, a]\!]^\Phi \longrightarrow_c \pi_r\, a'$ **:**
Similar to inductive case on pairs.

case $[\![\pi_r\, a]\!]^\Phi \equiv \pi_r\,(v_1, v_2) \longrightarrow_c v_r$ **:**
It follows that $[\![a]\!]^\Phi = (v_1, v_2)$. By $\mathsf{inf}^\circ_\mathsf{r}$ preserved by E_3 stripping (Lemma 31) a $\mathsf{inf}^\circ_\mathsf{r}$. By $[\![-]\!]^\Phi$ source-value property (Lemma 35) a $\mathsf{r'val}$. By $[\![-]\!]^\Phi$ outer value preservation (Lemma 36) there exists E_2, u_1, u_2 such that $a = E_2.(u_1, u_2)$. Thus $\pi_r\, a = \pi_r\, E_2.(u_1, u_2) \longrightarrow_{r'} E_2.u_r$. Note that $[\![a]\!]^\Phi = [\![E_2.(u_1, u_2)]\!]^\Phi = ([\![u_1]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi}, [\![u_2]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi}) = (v_1, v_2)$, thus $[\![E_2.u_r]\!]^\Phi = [\![u_r]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi} = v_r$ as required.

case $\lambda x{:}T.a$ **:**
Applying $[\![-]\!]^\Phi$ gives a function, and functions don't reduce.

case $a_1\, a_2$ **:**
Assume $\mathrm{wf}[a_1\, a_2] \wedge \mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi) \wedge (a_1, a_2)\ \mathsf{inf}^\circ_\mathsf{r} \wedge [\![a_1\, a_2]\!]^\Phi \longrightarrow_c e'$ and prove that there exists an a' such that $a_1\, a_2 \longrightarrow_{r'} a' \wedge e' = [\![a']\!]^\Phi$. We proceed by case split on the reductions of $[\![a_1\, a_2]\!]^\Phi$.

case $[\![a_1\, a_2]\!]^\Phi \longrightarrow_c e_1'\, [\![a_2]\!]^\Phi$ **:**
Similar to inductive case on pairs.

case $[\![a_1\, a_2]\!]^\Phi \longrightarrow_c [\![a_1]\!]^\Phi\, e_2'$ **:**
Similar to inductive case on pairs.

case $[\![a_1\, a_2]\!]^\Phi \equiv (\lambda x{:}T.e)\, v \longrightarrow_c \{v/x\}e$ **:**
Thus $[\![a_1]\!]^\Phi = \lambda x{:}T.e$ and $[\![a_2]\!]^\Phi = v$. By $\mathsf{inf}^\circ_\mathsf{r}$ preserved by E_3 stripping (Lemma 31) a_1 $\mathsf{inf}^\circ_\mathsf{r}$, so by $[\![-]\!]^\Phi$ source-value property (Lemma 35) a_1 $\mathsf{r'val}$. As a_1 $\mathsf{r'val}$ it follows by $\mathsf{inf}^\circ_\mathsf{r}$ preserved by E_3 stripping (Lemma 31) that a_2 $\mathsf{inf}^\circ_\mathsf{r}$, so by $[\![-]\!]^\Phi$ source-value property (Lemma 35) a_2 $\mathsf{r'val}$. By $[\![-]\!]^\Phi$ outer value preservation (Lemma 36) there exists $E_2, x, T, \hat{a}$ such that $a_1 = E_2.\lambda x{:}T.\hat{a}$. Thus, $(E_2.\lambda x{:}T.\hat{a})\, a_2 \longrightarrow_{r'} E_2.\mathbf{let}\ x = a_2\ \mathbf{in}\ \hat{a}$ and applying $[\![-]\!]^\Phi$ to the RHS gives $[\![\hat{a}]\!]^{\Phi'}$ where $\Phi' = \Phi, \mathcal{E}_c[E_2]^\Phi, x \mapsto [\![a_2]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi}$. We are left to show that $[\![\hat{a}]\!]^{\Phi'} = \{v/x\}e$. Do this by expanding $\{v/x\}e$

$$\begin{array}{rcll} \{v/x\}e & = & \{[\![a_2]\!]^\Phi/x\}[\![\hat{a}]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi, x \mapsto x} & \\ & = & [\![\hat{a}]\!]^{\Phi, \mathcal{E}_c[E_2]^\Phi, x \mapsto [\![a_2]\!]^\Phi} & (*) \\ & = & [\![\hat{a}]\!]^{\Phi'} & (**) \end{array}$$

(*) follows from $[\![-]\!]^-$ environment properties (i) (Lemma 26) and (**) is true as $\mathrm{fv}(a_2) \notin \mathrm{hb}(E_2)$.

case $\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2$ **:**
Assume $\mathrm{wf}[\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2]$; $\mathrm{fv}(\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2) \subseteq \mathrm{dom}(\Phi)$; $(\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2)\ \mathsf{inf}^\circ_\mathsf{r}$ and $[\![\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2]\!]^\Phi = [\![a_2]\!]^{\Phi_1} \longrightarrow_c e'$ where $\Phi_1 = \Phi, z \mapsto [\![a_1]\!]^\Phi$. By $\mathsf{inf}^\circ_\mathsf{r}$ preserved by E_3

stripping (Lemma 31) a_2 $\mathsf{inf}^\circ_\mathsf{r}$. By definition of wf[−] we have wf[a_2] $\wedge$ a_1 $\mathsf{r'val}$. By induction $a_2 \longrightarrow_{r'} a_2' \wedge e' = [\![a']\!]^{\Phi_1}$ (*), thus $\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2 \longrightarrow_{r'} \mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2'$. Now show that applying $[\![-]\!]^\Phi$ to the RHS of the previous transition gives e': $[\![\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2]\!]^\Phi = [\![a_2]\!]^{\Phi_1} = e'$ follows from (*).

case $\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2$ **:**

Assume wf[$\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2$]; fv($\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2$) $\subseteq$ dom(Φ); ($\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2$) $\mathsf{inf}^\circ_\mathsf{r}$ and $[\![\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2]\!]^\Phi = \mathbf{let}\ z = [\![a_1]\!]^\Phi\ \mathbf{in}\ [\![a_2]\!]^\Phi \longrightarrow_c e'$ (*). By wf[−] definition wf[a_1] $\wedge$ wf[a_2]. By $\mathsf{inf}^\circ_\mathsf{r}$ preserved by E_3 stripping (Lemma 31) a_1 $\mathsf{inf}^\circ_\mathsf{r}$.

We case split on the transitions of (*):

case $\mathbf{let}\ z = [\![a_1]\!]^\Phi\ \mathbf{in}\ [\![a_2]\!]^{\Phi,x\mapsto x} \longrightarrow_c \mathbf{let}\ z = e_1'\ \mathbf{in}\ [\![a_2]\!]^{\Phi,x\mapsto x}$ **:**

By induction $a_1 \longrightarrow_{r'} a_1' \wedge e_1' = [\![a_1']\!]^\Phi$. Thus $\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2 \longrightarrow_c \mathbf{let}_1\ z = a_1'\ \mathbf{in}\ a_2$ and we are left to show that applying $[\![-]\!]^-$ to the RHS of this results in the RHS of the case split:

$$\begin{array}{rcl} [\![\mathbf{let}_1\ z = a_1'\ \mathbf{in}\ a_2]\!]^\Phi & = & \mathbf{let}\ z = [\![a_1]\!]^\Phi\ \mathbf{in}\ [\![a_2]\!]^{\Phi,x\mapsto x} \\ & = & \mathbf{let}\ z = e_1'\ \mathbf{in}\ [\![a_2]\!]^{\Phi,x\mapsto x} \end{array}$$

as required.

case $\mathbf{let}\ z = [\![a_1]\!]\Phi\ \mathbf{in}\ [\![a_2]\!]^{\Phi,x\mapsto x} \longrightarrow_c \{[\![a_1]\!]^\Phi/z\}[\![a_2]\!]^{\Phi,x\mapsto x}$ **:**

Thus $[\![a_1]\!]^\Phi$ cval. By $[\![-]\!]^\Phi$ source-value property (Lemma 35) a_1 $\mathsf{r'val}$, thus $\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2 \longrightarrow_{r'} \mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2$. We are left to show that applying $[\![-]\!]^-$ to the RHS of this results in the RHS of the case split:

$$\begin{array}{rcl} [\![\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2]\!]^\Phi & = & \{[\![a_1]\!]^\Phi/z\}[\![a_2]\!]^{\Phi,x\mapsto x} \\ & = & [\![a_2]\!]^{\Phi,x\mapsto [\![a_1]\!]^\Phi} \end{array}$$

where the last step is valid by $[\![-]\!]^-$ environment properties (i) (Lemma 26) .

case Ω **:**

$[\![\Omega]\!]^\Phi = \Omega \longrightarrow_c \Omega$ and $\Omega \longrightarrow_{d'} \Omega$.

□

Lemma 52 (r'-r correspondence)

$$a\ \mathit{closed} \wedge \mathit{wf}[a] \wedge a \xrightarrow{l}_{r'} a' \wedge l \neq \mathit{zero} \implies \exists e'.\ \epsilon[a] \longrightarrow_r e' \wedge e' = \epsilon[a']$$

Proof We generalise to open terms and claim that it is sufficient to prove:

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

We prove this by induction on $a \xrightarrow{1}_{r'} a'$.

case (proj) **:**

Assume wf[$\pi_r\,(E_2.(u_1,u_2))$]. Then $\epsilon[\pi_r\,(E_2.(u_1,u_2))] = \pi_r\,\epsilon[E_2].(\epsilon[u_1],\epsilon[u_2])$. By $\epsilon[-]$ value preservation (Lemma 40) $\epsilon[u_1]$ rval and $\epsilon[u_2]$ rval. Thus by $\epsilon[-]$ preserves contexts (Lemma 42) $\pi_r\,\epsilon[E_2].(\epsilon[u_1],\epsilon[u_2]) \longrightarrow_r \epsilon[E_2].\epsilon[u_r] = \epsilon[E_2.u_r]$ as required.

case (app) **:**

Assume wf[$(E_2.\lambda x{:}T.\hat{a})\,u$]. Then $\epsilon[(E_2.\lambda x{:}T.\hat{a})\,u] = (\epsilon[E_2].\lambda x{:}T.\epsilon[\hat{a}])\,\epsilon[u]$. By $\epsilon[-]$ value preservation (Lemma 40) $\epsilon[u]$ rval, thus $\epsilon[E_2].((\lambda x{:}T.\epsilon[\hat{a}])\,\epsilon[u]) \longrightarrow_r \epsilon[E_2].\mathbf{let}\ x = \epsilon[u]\ \mathbf{in}\ \epsilon[\hat{a}]$. We are left to show that this is equal to the erasure of the RHS of the (app) reduction rule. Performing the erasure of the RHS we get $\epsilon[E_2.\mathbf{let}\ x = u\ \mathbf{in}\ \hat{a}] = \epsilon[E_2].\mathbf{let}\ x = \epsilon[u]\ \mathbf{in}\ \epsilon[\hat{a}]$, as required.

case (inst) **:**
Follow directly from inst match property (Lemma 46) .

case (zero) **:**
$l =$ zero.

case (cong) **:**
Assume wf$[E_3.a]$ and $a \longrightarrow_{r'} a'$. By well-formed properties (Lemma 24) wf$[a]$. By induction there exists an e' such that $\epsilon[a] \longrightarrow_r e' \wedge e' = \epsilon[a']$. We are now left to show that the erasure of $E_3.a$ reduces under λ_r to a term that is the erasure of $E_3.a'$. The following reasoning relies on the fact that $\epsilon[E_3]$ is a λ_r context, which can be established by $\epsilon[-]$ preserves contexts (Lemma 42) :

$$\begin{array}{rcl} \epsilon[E_3.a] & = & \epsilon[E_3].\epsilon[a] \\ & \longrightarrow_r & \epsilon[E_3].e' \\ & = & \epsilon[E_3].\epsilon[a'] \\ & = & \epsilon[E_3.a'] \end{array}$$

as required.

□

Lemma 53 (cr simulation) *R is a simulation from λ_c to λ_r*

Proof Recalling the definition of weak simulation and expanding the definition of R, we are required to prove

$$\begin{array}{l} (\exists a.\ \mathrm{wf}[a] \ \wedge\ a\ \mathsf{closed}\ \wedge\ e_1 = [\![\, a \,]\!]^{\varnothing}\ \wedge\ e_2 = \epsilon[a])\ \wedge\ e_1 \longrightarrow_c e_1' \implies \\ \qquad \exists e_2'.\ e_2 \longrightarrow_r^* e_2'\ \wedge\ (\exists a.\ \mathrm{wf}[a]\ \wedge\ a\ \mathsf{closed}\ \wedge\ e_1' = [\![\, a \,]\!]^{\varnothing}\ \wedge\ e_2' = \epsilon[a]) \end{array}$$

Assume

(13) $\exists a.\ \mathrm{wf}[a] \wedge a\ \mathsf{closed} \wedge e_1 = [\![\, a \,]\!]^{\varnothing} \wedge e_2 = \epsilon[a]$ and

(14) $e_1 \longrightarrow_c e_1'$.

Prove that there exists an e_2' such that

(15) $e_2 \longrightarrow_r^* e_2'$ and

(16) $\exists a.\ \mathrm{wf}[a] \wedge a\ \mathsf{closed} \wedge e_1' = [\![\, a \,]\!]^{\varnothing} \wedge e_2' = \epsilon[a]$

By $c{-}r'$ correspondence (lemma 51) there exists a' and a'' such that $a \xrightarrow{\text{insts}}{}^{*}_{r'} a'' \longrightarrow_{r'} a' \wedge a''\ \mathsf{inf_r} \wedge e_1' = [\![\, a' \,]\!]^{\varnothing}$. By inst match sequence (Lemma 47) there exists an e' such that $\epsilon[a] \xrightarrow{\text{insts}}{}^{*}_{r'} e' \wedge e' = \epsilon[a'']$.

We now case split on the reduction rule for $a'' \longrightarrow_{r'} a'$:

case $l =$ zero **:**
By $\epsilon[-]$ invariant under zeros (Lemma 38) we have $\epsilon[a''] = \epsilon[a']$, thus taking e_2' to be e' satisfies our proof obligation.

case otherwise **:**
By r'-r correspondence (Lemma 52) there exist e_2' such that $\epsilon[a''] \longrightarrow_r e_2' \wedge e_2' = \epsilon[a'']$.

□

B.2.2 c-r correspondence

Lemma 54 (r-r' correspondence)

$$a\ \textit{closed} \wedge \ \textit{wf}[a] \ \wedge \ \epsilon[a] \ \longrightarrow_r \ e' \implies \exists a', a''.\ a \xrightarrow{zero}{}^*_{r'} a'' \ \longrightarrow_{r'} \ a' \ \wedge \ a''\ \textit{znf}_r \ \wedge \ e' = \epsilon[a']$$

Proof We generalise to open terms and claim that it is sufficient to prove:

$$\mathrm{wf}[a] \ \wedge \ a\ \mathsf{znf}_\mathsf{r}^\circ \ \wedge \ \epsilon[a] \ \longrightarrow_r \ e' \implies \exists a'.\ a \ \longrightarrow_{r'} \ a' \ \wedge \ e' = \epsilon[a']$$

Let us show that this is sufficient. Suppose a closed; wf$[a]$; and $\epsilon[a] \ \longrightarrow_r \ e'$, then by ZNF reachability lemma (Lemma 39) there exists an a'' such that $a \xrightarrow{\text{zero}}{}^*_{r'} a'' \ \wedge \ a''\ \mathsf{znf}_\mathsf{r}$. As reduction can only reduce the number of free variables a'' closed and by reduction preserves well-formedness (Lemma 25) wf$[a'']$. It thus follows from our generalised claim that there exists an a' such that $a'' \ \longrightarrow_{r'} \ a' \ \wedge \ e' = \epsilon[a']$. The a' and a'' that we have demonstrated the existence of satisfy the conclusion of our original claim.

We prove the generalised claim by induction on a. The terms $z, ()$ and n are left unchanged by $\epsilon[-]$ and do not reduce under λ_r. The pair case is just application of the IH using well-formed properties (Lemma 24) and $\mathsf{znf}_\mathsf{r}^\circ$ preserved by E_3 stripping (Lemma 37) . The rest of the cases follow:

case $\pi_r\ a$ **:**
Assume wf$[\pi_r\ a]$; $\pi_r\ a\ \mathsf{znf}_\mathsf{r}^\circ$ and $\epsilon[\pi_r\ a] \ \longrightarrow_r \ e'$. By $\epsilon[-]$ source-value property (Lemma 43) a $\mathsf{r'val}$. By definition of wf$[-]$, wf$[a]$. By $\mathsf{znf}_\mathsf{r}^\circ$ preserved by E_3 stripping (Lemma 37) $a\ \mathsf{znf}_\mathsf{r}^\circ$. Observe $\epsilon[\pi_r\ a] = \pi_r\ \epsilon[a]$ and case split on the reductions of this:

case $\pi_r\ \epsilon[a] \ \longrightarrow_r \ \pi_r\ e'$ **:**
Thus $\epsilon[a] \ \longrightarrow_r \ e'$. By induction $a \ \longrightarrow_r \ a' \wedge e' = \epsilon[a']$, thus $\pi_r\ a \ \longrightarrow_{r'} \ \pi_r\ a' \wedge \pi_r\ e' = \epsilon[\pi_r\ a']$ as required.

case $\pi_r\ \epsilon[a] = \pi_r\ E_2.(v_1, v_2) \ \longrightarrow_r \ E_2.u_r$ **:**
By this case split $\epsilon[a] = E_2.(v_1, v_2)$ (*). By $\epsilon[-]$ outer value preservation (Lemma 44) there exists $\hat{E_2}, u_1, u_2$ such that $a = \hat{E_2}.(u_1, u_2)$ (**). Thus $\pi_r\ \hat{E_2}.(u_1, u_2) \ \longrightarrow_{r'} \ \hat{E_2}.u_r$. We are left to show that $\epsilon[\hat{E_2}.u_r] = E_2.v_r$. By (*) and (**) $\epsilon[\hat{E_2}] = E_2$ and $\epsilon[u_r] = v_r$, thus $\epsilon[\hat{E_2}.u_r] = \epsilon[\hat{E_2}].\epsilon[u_r] = E_2.v_r$ as required.

case $\lambda x{:}T.a$ **:**
$\epsilon[\lambda x{:}T.a] = \lambda x{:}T.\epsilon[a]$ which does not reduce under λ_r.

case $a_1\ a_2$ **:**
Assuming that (17) wf$[a_1\ a_2]$, (18) $a_1\ a_2\ \mathsf{znf}_\mathsf{r}^\circ$ and (19) $\epsilon[a_1\ a_2] \ \longrightarrow_r \ e'$, we can derive immediately (20) wf$[a_1] \ \wedge \ wf[a_2]$ and by (21) $a_1\ \mathsf{znf}_\mathsf{r}^\circ$.

we case split on the reduction of the last assumption:

case $\epsilon[a_1]\,\epsilon[a_2] \ \longrightarrow_r \ e_1'\,\epsilon[a_2]$ **:**
Inductive.

case $\epsilon[a_1]\,\epsilon[a_2] \ \longrightarrow_r \ \epsilon[a_1]\,e_2'$ **:**
Inductive.

case $\epsilon[a_1]\,\epsilon[a_2] \equiv (E_2.\lambda x{:}T.e)\ v \ \longrightarrow_r \ E_2.\mathbf{let}\ x = v\ \mathbf{in}\ e$ **:**
By well-formedness definition wf$[a_1] \ \wedge \ wf[a_2]$. By $\mathsf{znf}_\mathsf{r}^\circ$ preserved by E_3 stripping (Lemma 37) $a_1\ \mathsf{znf}_\mathsf{r}^\circ$. By $\epsilon[-]$ source-value property (Lemma 43) a_1 $\mathsf{r'val}$. By $\mathsf{znf}_\mathsf{r}^\circ$ preserved by E_3 stripping (Lemma 37) $a_2\ \mathsf{inf}_\mathsf{r}^\circ$. By $\epsilon[-]$ source-value property (Lemma 43) a_2 $\mathsf{r'val}$. By $\epsilon[-]$ outer value preservation (Lemma 44) a_1 is of the form $\hat{E_2}.\lambda x{:}T.a$.

First note that by alpha conversion we can ensure that $\mathrm{fv}(a_2) \notin \mathrm{hb}(\hat{E}_2)$. By case split $\epsilon[a_1] = \epsilon[\hat{E}_2.\lambda x{:}T.a] = \lambda x{:}T.\epsilon[a]$. By reduction rules $a_1\, a_2 = (\hat{E}_2.\lambda x{:}T.a)\, a_2 \longrightarrow_{r'} E_2.\mathbf{let}_0\ x = a_2\ \mathbf{in}\ a$. Then show that erasing this gives the desired result:

$$\epsilon[E_2.\mathbf{let}_0\ x = a_2\ \mathbf{in}\ a] = \mathbf{let}\ x = \epsilon[a_2]\ \mathbf{in}\ \epsilon[a]$$

We are left to show that $v = \epsilon[a_2]$, which is true by case split.

case $\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2$ **:**

This case proceeds by case analysis on the reductions of $\epsilon[\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2]$. There are two inductive cases, one in which $\epsilon[a_1]$ reduces and the other where $\epsilon[a_2]$ reduces. In both cases we use $\mathsf{znf}_\mathsf{r}^\circ$ preserved by E_3 stripping (Lemma 37) to establish open ZNF of a_1 or a_2 and then proceed by induction. The last possibility is for the term to reduce by doing an instantiation of z. In this case there exists E_3 such that $(\mathbf{let}\ z = u\ \mathbf{in}\ E_3.z) = \epsilon[\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2]$, and we are left to show that there exists an E_3' such that $a_2 = E_3'.z$, which is assured by $\epsilon[-]$ source context (Lemma 45).

case $\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2$ **:**

This case proceeds by case splitting on the reductions of $\epsilon[\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2]$. The first case is when $\epsilon[a_1]$ reduces, which goes by induction on a_1 after using $\mathsf{znf}_\mathsf{r}^\circ$ preserved by E_3 stripping (Lemma 37) to establish $a_1\ \mathsf{znf}_\mathsf{r}^\circ$. The other reduction is if a_1 is a value, then a zero reduction could occur, but this can not be the case as $\mathbf{let}_1\ z = a_1\ \mathbf{in}\ a_2$ is in open ZNF by assumption.

□

Lemma 55 (r'-c correspondence)

$$a\ \mathit{closed} \wedge \mathit{wf}[a] \wedge a \xrightarrow{l}_{r'} a' \wedge l \neq \mathit{inst} \implies \exists e'.\ [\![\, a \,]\!]^{\varnothing} \longrightarrow_c e' \wedge e' = [\![\, a' \,]\!]^{\varnothing}$$

Proof Generalising to open terms it is sufficient to prove:

$$\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi) \wedge \mathrm{wf}[a] \wedge a \xrightarrow{1}_{r'} a' \wedge l \neq \mathrm{insts} \implies \exists e'.\ [\![\, a \,]\!]^{\Phi} \longrightarrow_c e' \wedge e' = [\![\, a' \,]\!]^{\Phi}$$

This is true as if a closed then $\mathrm{fv}(a) = \varnothing \subseteq \mathrm{dom}(\Phi)$. We prove by induction on $a \xrightarrow{1}_{r'} a'$.

case (proj) **:**

Assume $\mathrm{fv}(\pi_r\,(E_2.(u_1,u_2))) \subseteq \mathrm{dom}(\Phi)$ and $\mathrm{wf}[\pi_r\,(E_2.(u_1,u_2))]$. Note that $[\![\, \pi_r\,(E_2.(u_1,u_2)) \,]\!]^{\Phi} = \pi_r\,([\![\, u_1 \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi}}, [\![\, u_2 \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi}})$ (*) and $[\![\, E_2.u_r \,]\!]^{\Phi} = [\![\, u_r \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi}}$. Our obligation is to show that (*) reduces to $[\![\, u_r \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi}}$.

From our assumptions we know $\mathrm{fv}(E_2.(u_1,u_2)) \subseteq \mathrm{dom}(\Phi)$ thus $\mathrm{fv}(E_2.u_r) \subseteq \mathrm{dom}(\Phi)$, moreover $\mathrm{fv}(u_r) \subseteq \mathrm{dom}(\Phi,\mathcal{E}_c[E_2]^{\Phi})$. By $[\![\, - \,]\!]^{-}$ value preservation (Lemma 28) $[\![\, u_r \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi}}$ cval. It follows that (*) reduces to $[\![\, u_r \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi}}$ under λ_c

case (app) **:**

Assume $\mathrm{fv}((E_2.\lambda x{:}T.a)\,u) \subseteq \mathrm{dom}(\Phi)$; $\mathrm{wf}[(E_2.\lambda x{:}T.a)\,u]$ and (22) $(E_2.\lambda x{:}T.a)\,u \longrightarrow_{r'} E_2.\mathbf{let}_0\ x = u\ \mathbf{in}\ a$.

Applying $[\![\, - \,]\!]^{\Phi}$ to the left-hand side of 22 and reduce.

$$\begin{array}{rcll} [\![\, (E_2.\lambda x{:}T.a)\,u \,]\!]^{\Phi} & = & (\lambda x{:}T.[\![\, a \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi},x\mapsto x})\,[\![\, u \,]\!]^{\Phi} & \\ & \longrightarrow_{r'} & \{[\![\, u \,]\!]^{\Phi}/x\}[\![\, a \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi},x\mapsto x} & \\ & = & [\![\, a \,]\!]^{\Phi,\mathcal{E}_c[E_2]^{\Phi},x\mapsto[\![\, u \,]\!]^{\Phi}} & (*) \end{array}$$

The last step uses $[\![-]\!]^-$ environment properties (i) (Lemma 26) . Now apply $[\![-]\!]$ to the right-hand side of 22 and show that it yields (*):

$$[\![\mathbf{let}_0\ x = u\ \mathbf{in}\ a]\!]^\Phi \quad = \quad [\![a]\!]^{\Phi, x \mapsto [\![u]\!]^\Phi}$$

case (inst) **:**

$l = \text{inst}$

case (zero) **:**

Follows directly from zero match property (Lemma 48) .

case (cong) **:**

Assuming $\text{fv}(E_3.a) \subseteq \text{dom}(\Phi)$; $\text{wf}[E_3.a]$; $E_3.a \ \longrightarrow_{r'}\ E_3.a'$ and $a \ \longrightarrow_{r'}\ a'$ we can deduce $\text{fv}(a) \subseteq \text{dom}(\Phi, \mathcal{E}_c[E_3]^\Phi)$, and $\text{wf}[a]$ by well-formed properties (Lemma 24) . Then by induction there exists e' such that $[\![a]\!]^{\mathcal{E}_c[E_3]^\Phi, \Phi} \ \longrightarrow_c\ e'$ and $e' = [\![a']\!]^{\Phi, \mathcal{E}_c[E_3]^\Phi}$. By $[\![-]\!]^-$ environment properties (i) (Lemma 26) this is the same as $[\![E_3.a]\!]^\Phi \ \longrightarrow_c\ e'$ and $e' = [\![E_3.a']\!]^\Phi$ as required.

□

Lemma 56 (rc simulation) *R is a weak simulation from λ_r to λ_c*

Proof Recalling the definition of weak simulation and expanding the definition of R, we are required to prove

$$\begin{aligned}&(\exists a.\ \text{wf}[a]\ \wedge\ a\ \mathsf{closed}\ \wedge\ e_1 = [\![a]\!]^\varnothing\ \wedge\ e_2 = \epsilon[a])\ \wedge\ e_2\ \longrightarrow_r\ e_2' \implies \\ &\quad \exists e_1'.\ e_1\ \longrightarrow_c^*\ e_1'\ \wedge\ (\exists a.\ \text{wf}[a]\ \wedge\ a\ \mathsf{closed}\ \wedge\ e_1' = [\![a]\!]^\varnothing\ \wedge\ e_2' = \epsilon[a])\end{aligned}$$

Assume

(23) $\exists a.\ \text{wf}[a]\ \wedge\ a\ \mathsf{closed}\ \wedge\ e_1 = [\![a]\!]^\varnothing\ \wedge\ e_2 = \epsilon[a]$ and

(24) $e_2\ \longrightarrow_r\ e_2'$.

Prove that there exists an e_1' such that

(25) $e_1\ \longrightarrow_c^*\ e_1'$ and

(26) $\exists a.\ \text{wf}[a]\ \wedge\ a\ \mathsf{closed}\ \wedge\ e_1' = [\![a]\!]^\varnothing\ \wedge\ e_2' = \epsilon[a]$

By r-r' correspondence (Lemma 54) there exists a' and a'' such that $a \xrightarrow{\text{zero}}{}^*_{r'} a''\ \longrightarrow_{r'}\ a'\ \wedge\ a''\ \mathsf{znf_r}\ \wedge$ $e_1' = \epsilon[a']$. By zero match sequence (Lemma 49) there exists an e' such that $[\![a]\!]^\varnothing \xrightarrow{\text{let}}{}^*_c e'\ \wedge\ e' = [\![a'']\!]^\varnothing$.

We now case split on the reduction rule for $a''\ \longrightarrow_{r'}\ a'$:

case $l = \text{inst}$ **:**

By $[\![-]\!]^-$ invariant under insts (Lemma 32) we have $[\![a'']\!]^\varnothing = [\![a']\!]^\varnothing$, thus taking e_1' to be e' satisfies our proof obligation.

case otherwise **:**

By r'-c correspondence (Lemma 55) there exist e'' such that $[\![a'']\!]^\varnothing\ \longrightarrow_c\ e_1'\ \wedge\ e_1' = [\![a'']\!]^\varnothing$, satisfying our proof obligation.

□

$$
\begin{tikzcd}
e \arrow[r, "*"', "r"] & v & \\
a \arrow[u, "\epsilon[-]"] \arrow[d, "\llbracket - \rrbracket^{\varnothing}"'] & a' \arrow[u, "\epsilon[-]"] \arrow[d, "\llbracket - \rrbracket^{\varnothing}"'] \arrow[r, "\text{zero}", "{}^{n}_{r'}"'] & u \arrow[ul, "\llbracket - \rrbracket^{\varnothing}"'] \arrow[d, "\epsilon[-]"] \\
e \arrow[r, "*", "c"'] & e'' \arrow[r, "\text{let}", "{}^{n}_{c}"'] & v'
\end{tikzcd}
$$

$$
\begin{tikzcd}
e \arrow[r, "*", "c"'] & v & \\
a \arrow[u, "\llbracket - \rrbracket^{\varnothing}"] \arrow[d, "\epsilon[-]"'] & a' \arrow[u, "\llbracket - \rrbracket^{\varnothing}"] \arrow[d, "\epsilon[-]"'] \arrow[r, "\text{inst}", "{}^{n}_{r'}"'] & u \arrow[ul, "\epsilon[-]"'] \arrow[d, "\llbracket - \rrbracket^{\varnothing}"] \\
e \arrow[r, "*", "r"'] & e'' \arrow[r, "\text{inst}", "{}^{n}_{c}"'] & v'
\end{tikzcd}
$$

Figure 21: Operational reasoning of r-c equivalence

B.3 Equivalence

Having demonstrated a bisimulation between λ_c and λ_r we must show that the termination of expressions coincides for both systems in order to show that the two are observationally equivalent. Figure 21 shows diagrammatically how the proof of the main theorem will proceed. First we establish some facts about types and the auxiliary operations.

Definition 17 (environment-substitution correspondence)

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

Definition 18 (equality on λ terms up to functions) We define $=_\lambda$ to be the standard equality relation up to alpha-equivalence, but extended to equate every function.

Lemma 57 (value correspondence) *if* $\Phi = \Phi_k$ *where*

$$
\begin{array}{lcl}
\Phi_0 & = & \varnothing \\
\Phi_{n+1} & = & \Phi_n, x_{n+1} \mapsto \llbracket u_{n+1} \rrbracket^{\Phi_n} \quad \textit{where } fv(\llbracket u_{n+1} \rrbracket^{\Phi_n}) = \varnothing
\end{array}
$$

and $fv(u) \subseteq dom(\Phi)$ *and* $wf[u]$ *then* $S[\Phi]\{\!| \epsilon[u] |\!\} =_\lambda \llbracket u \rrbracket^{\Phi}$

Proof

The proof proceeds by induction on the structure of u.

case n; () **:**
Immediate.

case (u_1, u_2) **:**
Assume $\mathrm{wf}[(u_1,u_2)]$; $\mathrm{fv}((u_1,u_2)) \subseteq \mathrm{dom}(\Phi)$. It can easily be verified that $\mathrm{wf}[u_1] \wedge \mathrm{wf}[u_2]$, $\mathrm{fv}(u_1) \subseteq \mathrm{dom}(\Phi) \wedge \mathrm{fv}(u_2) \subseteq \mathrm{dom}(\Phi)$. By induction on u_1 we have $\mathrm{S}[\Phi]\{\mid \epsilon[u_1] \mid\} =_\lambda [\![u_1]\!]^\Phi$ and similarly by induction on u_2 we have $\mathrm{S}[\Phi]\{\mid \epsilon[u_2] \mid\} =_\lambda [\![u_2]\!]^\Phi$. It follows that

$$\begin{array}{rll} [\![(u_1,u_2)]\!]^\Phi & =_\lambda & ([\![u_1]\!]^\Phi, [\![u_2]\!]^\Phi) \\ & =_\lambda & (\mathrm{S}[\Phi]\{\mid \epsilon[u_1] \mid\}, \mathrm{S}[\Phi]\{\mid \epsilon[u_2] \mid\}) \\ & =_\lambda & \mathrm{S}[\Phi]\{\mid (\epsilon[u_1], \epsilon[u_2]) \mid\} \end{array}$$

as required.

case $\lambda x{:}T.a$ **:**
$\sigma\{\mid \epsilon[\lambda x{:}T.a] \mid\} = \lambda x{:}T.\sigma\epsilon[a]$ and $[\![\lambda x{:}T.a]\!]^\Phi = \lambda x{:}T.[\![a]\!]^{\Phi, x \mapsto x}$ which are both functions and therefore are equated by $=_\lambda$.

case $\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2$ **:**
Assume $\mathrm{wf}[\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2]$ and $\mathrm{fv}(\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2) \subseteq \mathrm{dom}(\Phi)$. We are required to prove

$$\mathrm{S}[\Phi]\{\mid \epsilon[\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2] \mid\} =_\lambda [\![\mathbf{let}_0\ z = u_1\ \mathbf{in}\ u_2]\!]^\Phi$$

which holds if and only if

$$\mathrm{S}[\Phi, z \mapsto \{\mid \epsilon[u_1] \mid\}]\{\mid \epsilon[u_2] \mid\} =_\lambda [\![u_2]\!]^{\Phi, z \mapsto [\![u_1]\!]^\Phi} \quad (*)$$

From our initial assumptions it is clear that $\mathrm{fv}(u_1) \subseteq \mathrm{dom}(\Phi)$ and all of the values in the domain of Φ are closed. It follows by a simple induction (proving $\mathrm{fv}(a) \subseteq \mathrm{dom}(\Phi) \implies \mathrm{fv}([\![a]\!]^\Phi) \subseteq \mathrm{fv}(\Phi)$) that $\mathrm{fv}([\![u_1]\!]^\Phi) = \varnothing$. It then follows by induction that (*) holds, as required.

□

Lemma 58 (typing is substitutive)

$$\Phi \vdash v{:}T \ \wedge\ \Phi, z{:}T \vdash e{:}T' \implies \Phi \vdash \{v/z\}e{:}T'$$

Proof Prove by induction on $\Phi \vdash e{:}T'$:

case (var) **:**
Assume $\Phi \vdash v{:}T \wedge \Phi, z{:}T \vdash x{:}T'$ (*) and prove $\Phi \vdash \{v/z\}x{:}T'$. If $z = x$ then $T = T'$ and we are required to show $\Phi \vdash v{:}T'$, which is assured by assumption. If $z \neq x$ then we must show $\Phi \vdash x{:}T'$ (**). Seeing as $z \neq x$ and (*) holds, then $x{:}T' \in \Phi$, therefore (**) holds as required.

case (int); (unit) **:**
Trivial.

case (fun) **:**
Assume $\Phi \vdash v{:}T$; $\Phi, z{:}T \vdash \lambda x{:}T'.e'{:}T' \to T''$ and $\Phi, z{:}T, x{:}T' \vdash x{:}T''$. By alpha conversion $x \neq z$. By Permutation Lemma (Lemma [17]) $\Phi, z{:}T, x{:}T' \vdash x{:}T''$. By induction $\Phi, x{:}T' \vdash \{v/z\}e'{:}T''$. Thus by typing rules $\Phi \vdash \lambda x{:}T'.\{v/z\}e'{:}T' \to T''$ and as $x \neq z$ we have $\Phi \vdash \{v/z\}(\lambda x{:}T'.e'){:}T' \to T''$ as required.

case (app); (proj); (pair) **:**
Inductive.

case (let) **:**
Assume $\Phi \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2{:}T$; $\Phi \vdash e_1{:}T_1$ and $\Phi, x{:}T_1 \vdash e_2{:}T_2$. By alpha conversion $x \neq z$. By induction $\Phi \vdash \{v/z\}e_1{:}T_1$ and $\Phi, x_1{:}T_1 \vdash \{v/z\}e_2{:}T_2$. Result follows by typing rules.

□

Lemma 59 ($\{\!|\ \_\ |\!\}$ type preservation)

$$\Phi \vdash u{:}T \implies \Phi \vdash \{\!|\ u\ |\!\}{:}T$$

Proof Prove by induction on $\Phi \vdash u{:}T$:

case (int); (unit) **:**
$\{\!|\ n\ |\!\} = n$ and $\{\!|\ ()\ |\!\} = ()$.

case (fun) **:**
Assuming $\Phi \vdash \lambda x{:}T'.e$ (*) and $x{:}T', \Phi \vdash e{:}T$. Now $\{\!|\ \lambda x{:}T'.e\ |\!\} = \lambda x{:}T'.e$ so we are done by (*).

case (app); (proj) **:**
Terms not values.

case (pair) **:**
Assume $\Phi \vdash (u_1, u_2){:}T_1 * T_2$; $\Phi \vdash u_1{:}T_1$ and $\Phi \vdash u_2{:}T_2$. By induction $\Phi \vdash \{\!|\ u_1\ |\!\}$ and by induction again $\Phi \vdash \{\!|\ u_2\ |\!\}$. Thus $\Phi \vdash \{\!|\ (u_1, u_2)\ |\!\}{:}T_1 * T_2$.

case (let) **:**
Assuming $\Phi \vdash \textbf{let}\ x = u_1\ \textbf{in}\ u_2{:}T$ we have $\{\!|\ \textbf{let}\ x = u_1\ \textbf{in}\ u_2\ |\!\} = \{\{\!|\ u_1\ |\!\}/x\}\{\!|\ u_2\ |\!\}$ and thus by typing is substitutive (Lemma 58) $\{\Phi \vdash \{\!|\ u_1\ |\!\}/x\}\{\!|\ u_2\ |\!\}$ as required.

□

We now prove theorem 23:

Proof We begin by proving point 1 of the theorem.

First prove:

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

Assume e closed and $e \longrightarrow_c^* v_1$, and recall eRe by $\mathrm{id}_\lambda \subseteq R$ (Lemma 50) . By c-r simulation (Lemma 53) 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 assured that

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

We are left to show $e' \longrightarrow_r^* e''$ and e'' rval. By $\epsilon[-]$ source-value property (Lemma 43) it suffices to prove that there exists an a' such that a' r'val $\wedge$ wf$[a']$ $\wedge$ a' $\mathsf{znf_r}$ $\wedge$ $e'' = \epsilon[a']$.

Suppose that a $\mathsf{inf_r}$, then by $[\![-]\!]^{\Phi}$ source-value property (Lemma 35) a r'val. By $\epsilon[-]$ value preservation (Lemma 40) $\epsilon[a]$ rval as required.

Now suppose that $\neg(a\ \mathsf{inf_r})$ then by INF reachability lemma (Lemma 34) there exists an a'' such that $a \longrightarrow_{r'}^* a'\ \wedge\ a'\ \mathsf{inf_r}$. By reduction preserves well-formedness (Lemma 25) wf$[a']$ and by $[\![-]\!]^{-}$

invariant under insts (Lemma 32) $v_1 = [\![a']\!]^{\varnothing}$. Thus by $[\![-]\!]^{\Phi}$ source-value property (Lemma 35) a' $\mathsf{r'val}$. By inst match sequence (Lemma 47) there exists an e'' such that $e' \longrightarrow_r^* e'' \wedge e'' = \epsilon[a']$ as required.

Now prove the main theorem:

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

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 $e \longrightarrow_r^* v_2 \wedge \mathrm{wf}[u] \wedge u$ closed $\wedge\ n = [\![u]\!]^{\varnothing} \wedge v_2 = \epsilon[u]$.

We are left to show that $n = \{\!| v_2 |\!\}$. By value correspondence (Lemma 57) $\{\!| \epsilon[u] |\!\} = [\![u]\!]^{\varnothing}$. 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 λ_r (Lemma [20]) $\vdash v_2{:}\mathsf{int}$, thus $\vdash \epsilon[u]{:}\mathsf{int}$ by dint of equality with v_2. By $\{\!| \_ |\!\}$ type preservation (Lemma 59) $\vdash \{\!| \epsilon[u] |\!\}{:}\mathsf{int}$, as required.

Now prove point 2.

First prove:

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

Assume e closed and $e \longrightarrow_r^* v_1$, and recall eRe by $\mathrm{id}_\lambda \subseteq R$ (Lemma 50) . By r-c simulation (Lemma 56) R is a r-c simulation, thus there exists an e' such that $e \longrightarrow_c^* e'$ and $e'Rv_1$. Expanding the definition of R in the latter, we are assured that

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

We are left to show $e' \longrightarrow_c^* e''$ and e'' cval. By $[\![-]\!]^{\Phi}$ source-value property (Lemma 35) it suffices to prove that there exists an a' such that a' $\mathsf{r'val}$ $\wedge\ \mathrm{wf}[a'] \wedge a'$ $\mathsf{inf_r}$ $\wedge\ e'' = [\![a']\!]^{\varnothing}$.

Suppose that a $\mathsf{znf_r}$ then by $\epsilon[-]$ source-value property (Lemma 43) a $\mathsf{r'val}$. By $[\![-]\!]^{-}$ value preservation (Lemma 28) $[\![a']\!]^{\varnothing}$ cval as required.

Now suppose that $\neg(a\ \mathsf{znf_r})$ then by ZNF reachability lemma (Lemma 39) there exists an a'' such that $a \xrightarrow{\text{zeros}}{}^*_{r'} a' \wedge a'$ $\mathsf{znf_r}$. By reduction preserves well-formedness (Lemma 25) $\mathrm{wf}[a']$ and by $\epsilon[-]$ invariant under zeros (Lemma 38) $v_1 = \epsilon[a']$. Thus by $\epsilon[-]$ source-value property (Lemma 43) a' $\mathsf{r'val}$. By zero match sequence (Lemma 49) there exists an e'' such that $e' \longrightarrow_c^* e'' \wedge e'' = \epsilon[a']$ as required.

Now prove the main theorem:

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

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

We are left to show that $\{\!| u |\!\} = n$. By value correspondence (Lemma 57) $\{\!| \epsilon[u] |\!\} = [\![u]\!]^{\varnothing}$. 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 λ_r (Lemma [20]) $\vdash v{:}\mathsf{int}$, thus $\vdash \epsilon[u]{:}\mathsf{int}$ by dint of equality with v. By $\{\!| \_ |\!\}$ type preservation (Lemma 59) $\vdash \{\!| \epsilon[u] |\!\}{:}\mathsf{int}$, as required.

□

B.4 Observational equivalence between λ_d and λ_c

The goal of this section is to prove the observational equivalence between λ_d and λ_c, as stated in the following theorem:

Theorem 60 *For all $e \in \lambda$ the following hold:*

1. $\vdash e{:}\mathsf{int} \implies (e \longrightarrow_c^* n \implies \exists v.\ e \longrightarrow_d^* v \ \wedge\ n = \{\mid v \mid\})$
2. $\vdash e{:}\mathsf{int} \implies (e \longrightarrow_d^* v \implies \exists n.\ e \longrightarrow_c^* n \ \wedge\ n = \{\mid v \mid\})$

This proof follows the same structure as that of the observational equivalence proof between λ_r and λ_c. We borrow concepts and definitions from this earlier proof, and only highlight the differences in proofs which follow a similar structure to their counterparts in the λ_r proof.

We borrow the annotated syntax λ'; the functions $\iota[-]$, $\epsilon[-]$, $\mathrm{bc}(-)$ and $\mathcal{E}_c[-]^-$; and the predicate $\mathrm{wf}[-]$ from the λ_r proof.

As we are ultimately interested only in closed terms, we are free to alter the behaviour of λ_c on open terms so long as it remains the same when restricted to closed terms. We do this by adding identifiers to the set of values for λ_c:

$$v ::= z \mid n \mid () \mid \lambda x{:}T.e$$

Definition 19 ($\lambda_{d'}$) This is as defined for $\lambda_{r'}$ except we add *destruct contexts*:

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

and we replace the (inst) reduction rule with two instantiation rules:

$$\begin{array}{llll} \text{(inst-1)} & \mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.z & \longrightarrow & \mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.u \\ \text{(inst-2)} & R.E_2.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_2'.z & \longrightarrow & R.E_2.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_2'.u \end{array}$$

Definition 20 (Environment)

An *environment* Φ is a list containing pairs whose first component is an identifier and whose second component is a c-value. Environments have the property that $\forall x \in \mathrm{dom}(\Phi).\ \Phi(x) = v \ \wedge\ \forall z \in \mathrm{fv}(v).z {\leq_\Phi} x$ where $\leq_\Phi$ is the ordering of the identifiers in Φ. In addition we require that all the first components of the pairs in the list are disjoint. We write $\Phi, z \mapsto v$ for the disjoint extension of Φ forming a new environment. We write $\Phi[z \mapsto v]$ for the environment acting as Φ, but mapping z to v

Definition 21 ($\llbracket - \rrbracket^-$) We use the definition from the λ_r case with the following change:

$$\llbracket z \rrbracket^\Phi = \Phi^*(z)$$

where we define Φ^* as the least fixpoint of the monotone operator F:

$$F(\Phi) = \Phi[x \mapsto z \mid \exists y.\ \Phi(x) = y \ \wedge\ \Phi(y) = z]$$

Definition 22 (Instantiation normal form (INF)) A term a is in *instantiation normal form* (INF) if and only if there does not exist an a' such that $a \xrightarrow{\text{inst}}_{d'} a'$, where inst is inst-1 or inst-2. We write $a\ \mathsf{inf_d}$ when a is in INF.

Definition 23 (open INF) A possibly open term a is in *open instantiation normal form* if and only if there does not exist an E_3, R, E_2 and z such that $a = E_3.R.E_2.z$. We write $a\ \mathsf{inf_d^\circ}$ when a is in open INF.

Transforming proofs from λ_r to λ_d In order to avoid duplicating tedious proofs, we would like to reuse as much reasoning from the λ_r proof as possible. To do this we will enumerate the entities we have changed in the setup above to guide the reader, informally, in how λ_r proofs were transformed into a corresponding λ_d one.

The entities we changed are:

- added identifiers to values
- added destruct contexts R;
- changed the (inst) rule;
- changed the environment, Φ, and $[\![-]\!]^{-}$.

We notice that every R context is an A_1 context. In particular this means that the $E_3.R.E_2$ context in the new (inst) rule is a particular form of E_3 context.

Although we have changed the environment, we have weakened the conditions for adding elements to it; while when we use elements from it they are taken out of Φ^*, the "transitive closure" of Φ, which is an environment in the sense of that used for the λ_r proof.

Thus, informally, a proof in the λ_r equivalence result will remain a valid proof, or have a trivial translation, in the λ_d equivalence result if the following conditions hold:

1. the proof does not rely on the form of values;
2. the proof does not rely on the form of an E_3 context;
3. the proof does not rely on the actual elements in the codomain of the environment.

If these properties hold of a proof in the λ_r case then we will say that the proof of the property follows *directly from the argument given in the λ_r case.* If this is the case, then the lemma is stated without proof.

Lemma 61 (well-formed properties)

(i) $\mathit{wf}[E_3.a] \iff \mathit{wf}[E_3] \wedge \mathit{wf}[a]$

(ii) $\mathit{wf}[a] \wedge a \longrightarrow_{d'} a' \implies \mathit{wf}[a']$

Proof (i) follows directly from the λ_r case. (ii) The proof is by induction on $a \longrightarrow_{d'} a'$. All the common cases follow analogously from the λ_r proof, then we are left with the two inst cases, which are similar. We give (inst-1): assume wf[$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.z$] then by (i) wf[$E_3.R.E_2.z$] and by definition of well-founded wf[u], thus by (i) wf[$E_3.R.E_2.u$]. It follows that wf[$\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.u$] as required. □

Lemma 62 ($[\![-]\!]^{-}$ environment properties)

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

(ii) If $x \notin \mathit{fv}(a)$ *then* $[\![a]\!]^{\Phi,x\mapsto v} = [\![a]\!]^{\Phi}$

Proof Prove (i) by induction on a. The interesting case is the identifier case:

case z **:**

Assume wf$[z]$; $z \in \mathrm{dom}(\Phi)$ and $\mathrm{fv}(v) \subseteq \mathrm{dom}(\Phi)$. It suffices to prove $\{v/x\}[\Phi, x \mapsto x]^*(z) = [\Phi, x \mapsto v]^*(z)$. There are three cases to consider: $z = x$; $z \neq x \ \wedge\ \Phi^*(z) = z$; and $z \neq x \ \wedge\ \Phi^*(z) = v'$ where v' is not an identifier. In the first and second cases are trivial, so lets consider the last. First lets point out that $[\Phi, x \mapsto x]^*(z) = [\Phi, x \mapsto v]^*(z) = \Phi^*(z)$ as x cannot appear free in $\mathrm{cod}(\Phi)$. Thus it is sufficient to show that $x \notin \mathrm{fv}(v')$ as then $\{v/x\}v' = v'$. To show this, note that every environment has a unique domain, therefore as $\Phi, x \mapsto v$ is an environment $x \notin \mathrm{dom}(\Phi)$. Furthermore, by the constraint on free variables $x \notin \mathrm{fv}(\mathrm{cod}(\Phi))$ from which it follows that $x \notin \mathrm{fv}(\mathrm{cod}(\Phi^*))$, thus $x \notin \mathrm{fv}(v')$.

□

Lemma 63 (environments over contexts) $\mathit{fv}(E_3.a) \subseteq \Phi \iff \mathit{fv}(a) \subseteq (\Phi, \mathcal{E}_c[E_3]^{\Phi})$

Lemma 64 ($[\![-]\!]^-$ value preservation)

$$\mathit{fv}(u) \subseteq \mathit{dom}(\Phi) \ \wedge\ \mathit{wf}[u] \implies [\![u]\!]^{\Phi}\ \mathsf{cval}$$

Proof The proof is similar to the λ_r one, the new case is identifiers: as $\mathrm{fv}(z) \subseteq \mathrm{dom}(\Phi)$ we have $[\![z]\!]^{\Phi} = \Phi^*(z)$ which by definition is a c-value. □

Lemma 65 ($[\![-]\!]$ distributes over contexts) *For all E_3, Φ and a, if $\mathit{fv}(a) \subseteq \mathit{dom}(\Phi)$ and $\mathit{wf}[E_3.a]$ then $[\![E_3.a]\!]^{\Phi} = [\![E_3]\!]^{\Phi}.[\![a]\!]^{\Phi,\mathcal{E}_c[\Phi]^{E_3}}$*

Lemma 66 ($[\![-]\!]$ preserves contexts) *If $\mathit{fv}(E_3) \subseteq \mathit{dom}(\Phi)$ and $\mathit{wf}[E_3]$ then there exists a λ_c reduction context E such that $[\![E_3]\!]^{\Phi} = E$.*

Proof Follows the λ_r proof as Lemma 64 holds. □

We now show that there are only ever finitely many instantiation steps to the next instantiation normal form. We first make some observations to motivate our approach:

- Every evaluation context E_3 describes a tree with a unique hole
- The path in this tree from the hole to the root is unique

Definition 24 (Derived before order) Every evaluation context E_3 induces a *derived before* total order on the variables bound along the path from the hole to the root such that $z \lhd_{E_3} z'$ if and only if z is closer to the root than z'.

Definition 25 (Instantiation chain) A sequence $x_1, x_2, \ldots$ is called an *instantiation chain* for an evaluation context E_3 if and only if $x_i \lhd x_j$ whenever $i < j$.

Lemma 67 $E_3.z \xrightarrow{\mathit{inst}}_{d'} E_3.z' \implies z' \lhd z$

Proof Prove $z \lhd z'$ under the assumption that $E_3.z \longrightarrow_{d'} E_3.z'$. In both of the let rules, the syntax ensures that $z' \lhd z$ □

Lemma 68 $\mathit{wf}[E_3.z] \ \wedge\ E_3.z \xrightarrow{\mathit{inst}}_{d'} E_3.u \ \wedge\ u \neq z' \implies E_3.u\ \mathsf{inf}^{\circ}_{\mathsf{d}}$

Proof We assume that $E_3.z \xrightarrow{\mathrm{inst}}_{d'} E_3.u \ \wedge\ u \neq z'$ and proceed by case analysis on the inst transition.

case (inst-1) **:**

We have $E_3.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3'.R.E_2.z \longrightarrow_{d'} E_3.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3'.R.E_2.u$ with the side conditions that $z \notin \mathrm{hb}(E_3', E_2)$ and $\mathrm{fv}(u) \notin z, \mathrm{hb}(E_3', E_2)$. Again, there are two possibilities depending on R:

- $E_3.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3'.\pi_r\ .E_2.u$

- $E_3.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3'.(\_u).E_2.u$

  Take possibility 1: we can rewrite as $E_3.\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3'.\pi_r\ (E_2.u)$, which is either stuck or if u is a pair can reduce via (proj), in either case the term is in INF. Possibility 2 is similar.

case (inst-2) **:**

Similar.

□

Before the next lemma, we note that $E_3.z$ where z does not bind around the hole in E_3 is in instantiation normal form as no more reductions can be done.

Lemma 69 *For all closed a, there exists an a' such that $a \xrightarrow{inst}{}^{*}_{d'} a'$ and a' $\mathsf{inf_d}$*

Proof Either a can do an inst or it can not. If it can not then it must be in instantiation normal form for λ_d, so suppose that $a \xrightarrow{\text{inst}}_{d'} a''$, then a has the general form $E_3.z$ and a'' the general form $E_3.u$. Either u is a non-identifier value or it is an identifier. In the former case lemma 68 holds and $E_3.u$ $\mathsf{inf_d^\circ}$. In the latter case, lemma 67 tells us that this can result in at most finitely many instantiations before the instantiation is not a bound identifier, at which point it must be a non-identifier value, or a free variable, either way we are in instantiation normal form for closed terms.

□

Lemma 70 (INF preserved by E_3 stripping) *For any evaluation context E_3, $E_3.a$ $\mathsf{inf_d^\circ} \implies a$ $\mathsf{inf_d^\circ}$*

Lemma 71 ($[\![-]\!]^{-}$ invariant under insts) $wf[a] \ \wedge\ fv(a) \subseteq dom(\Phi) \ \wedge\ a \xrightarrow{insts}{}^{*}_{d'} a' \implies [\![a]\!]^{\Phi} = [\![a']\!]^{\Phi}$

Proof We first prove the single step case by induction on $a \xrightarrow{\text{inst}}_{d'} a'$:

case (inst-1) **:**

$$\begin{array}{rcll}
[\![\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.z]\!]^{\Phi} & = & [\![z]\!]^{\Phi, z \mapsto [\![u]\!]^{\Phi}, \mathcal{E}_c[E_3.R.E_2]^{\Phi}} & \\
& = & [\Phi, z \mapsto [\![u]\!]^{\Phi}, \mathcal{E}_c[E_3.R.E_2]^{\Phi}]^{*}(z) & \\
& = & [\![u]\!]^{\Phi} & \\
& = & [\![u]\!]^{\Phi, z \mapsto [\![u]\!]^{\Phi}, \mathcal{E}_c[E_3.R.E_2]^{\Phi}} & (*) \\
& = & [\![\mathbf{let}_0\ z = u\ \mathbf{in}\ E_3.R.E_2.u]\!]^{\Phi} &
\end{array}$$

Where (*) is valid by part (ii) of Lemma 62

case (inst-2) **:**

Similar to the previous case.

case (cong) **:**

Assume wf$[E_3.a]$ and fv$(E_3.a) \subseteq$ dom(Φ). By Lemma 61wf$[a]$. Let $\Phi' = \Phi, \mathcal{E}_c[E_3]^{\Phi}$, then fv$(a) \subseteq$ dom(Φ'). By induction $[\![a]\!]^{\Phi'} = [\![a']\!]^{\Phi'}$ (*). Now $[\![E_3.a]\!]^{\Phi} = [\![a]\!]^{\Phi'}$ and $[\![E_3.a']\!]^{\Phi} = [\![a']\!]^{\Phi'}$, thus by (*) we are done.

□

Lemma 72 ($[\![-]\!]^{\Phi}$ source-value property) *For all $\lambda_{d'}$ expressions a, the following holds:*

$$wf[a] \ \wedge\ a\ \mathsf{inf_r^\circ} \ \wedge\ fv(a) \subseteq dom(\Phi) \ \wedge\ [\![a]\!]^{\Phi}\ \mathsf{cval} \implies a\ \mathsf{d'val}$$

Proof This proof is the same apart from the identifier case, which is immediate as identifiers are values. $\square$

Notice in the next lemma that an extra restriction is needed when compared to the corresponding λ_r lemma, that is, the value u must not be an identifier.

Lemma 73 ($\llbracket - \rrbracket$ outer value preservation) *For all $\lambda_{d'}$ values u that are not identifiers:*

(a) If $wf[u]$, $fv(a) \subseteq dom(\Phi)$ and $\llbracket u \rrbracket^\Phi = \lambda x{:}T.e$ then there exists E_2, a, z such that $u = E_2.\lambda x{:}T.a$

(b) $\llbracket u \rrbracket^\Phi = (v_1, v_2) \implies \exists E_2, u_1, u_2.\ u = E_2.(u_1, u_2)$

Lemma 74 ($\mathsf{znf}^\circ_\mathsf{d}$ preserved by E_3 stripping) $E_3.a\ \mathit{znf}^\circ_d \implies a\ \mathit{znf}^\circ_d$

Lemma 75 ($\epsilon[-]$ invariant under zeros) $wf[a] \ \wedge\ a \xrightarrow{zeros}{}^*_{r'} a' \implies \epsilon[a] = \epsilon[a']$

Lemma 76 (ZNF reachability) *For all closed a, if $wf[a]$ then there exists a' such that $a \xrightarrow{zero}{}^*_{d'} a' \ \wedge$ $a'\ \mathit{znf}_r$*

Lemma 77 ($\epsilon[-]$ value preservation)

$$wf[u] \implies \epsilon[u]\ \mathit{dval}$$

Lemma 78 ($\epsilon[-]$ distributes over contexts) $\epsilon[E_3.a] = \epsilon[E_3].\epsilon[a]$

Lemma 79 ($\epsilon[-]$ preserves contexts) *If $wf[E_3]$ then there exists a λ_r reduction context E'_3 such that $\epsilon[E_3] = E'_3$.*

Lemma 80 ($\epsilon[-]$ source-value property) $wf[a] \ \wedge\ a\ \mathit{znf}^\circ_d \ \wedge\ \epsilon[a]\ \mathit{dval} \implies a\ \mathit{d'val}$

Lemma 81 ($\epsilon[-]$ outer value preservation) *For all $\lambda_{d'}$ values u:*

(a) If $wf[u]$ and $\epsilon[u] = E_2.\lambda x{:}T.e$ then there exists $\hat{E}_2, a, z$ such that one of the following holds:

(i) $u = \hat{E}_2.\lambda x{:}T.a$

(b) $\epsilon[u] = E_2.(v_1, v_2) \implies \exists \hat{E}_2, u_1, u_2.\ u = \hat{E}_2.(u_1, u_2)$

Lemma 82 ($\epsilon[-]$ source context) *If $\epsilon[a] = E_3.e$ and $a\ \mathit{znf}^\circ_d$ 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 83 (inst match property)

$$wf[a] \ \wedge\ a \xrightarrow{inst}_{d'} a' \implies \exists e'.\ \epsilon[a] \xrightarrow{inst}_d e' \ \wedge\ e' = \epsilon[a']$$

Lemma 84 (inst match sequence)

$$wf[a] \ \wedge\ a \xrightarrow{inst}{}^n_{d'} a' \implies \exists e'.\ \epsilon[a] \xrightarrow{inst}{}^n_d e' \ \wedge\ e' = \epsilon[a']$$

Lemma 85 (zero match property)

$$wf[a] \ \wedge\ fv(a) \subseteq dom(\Phi) \ \wedge\ a \xrightarrow{zero}_{d'} a' \implies \exists e'.\ \llbracket a \rrbracket^\Phi \xrightarrow{let}_c e' \ \wedge\ e' = \llbracket a' \rrbracket^\Phi$$

Lemma 86 (rec-zero match sequence)

$$wf[a] \ \wedge\ fv(a) \subseteq dom(\Phi) \ \wedge\ a \xrightarrow{zero}{}^{n}_{d'} a' \implies \exists e'.\ \llbracket a \rrbracket^{\Phi} \xrightarrow{let}{}^{n}_{c} e' \ \wedge\ e' = \llbracket a' \rrbracket^{\Phi}$$

Definition 26 (Candidate bisimulation)

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

Lemma 87 ($\mathbf{id}_\lambda \subseteq R$) *The candidate bisimulation R contains id_λ.*

Lemma 88 (c-d' correspondence)

$$a \ \mathit{closed} \wedge\ wf[a] \ \wedge\ \llbracket a \rrbracket^{\varnothing} \longrightarrow_c e' \implies \exists a', a''.\ a \xrightarrow{inst}{}^{*}_{d'} a'' \longrightarrow_{d'} a' \ \wedge\ a'' \ \mathit{inf_d} \wedge\ e' = \llbracket a' \rrbracket^{\varnothing}$$

Proof We prove the generalised statement:

$$\mathrm{wf}[a] \ \wedge\ a \ \mathsf{inf}^{\circ}_{\mathsf{d}} \ \wedge\ \llbracket a \rrbracket^{\Phi} \longrightarrow_c e' \implies \exists a'.\ a \longrightarrow_{d'} a' \ \wedge\ e' = \llbracket a' \rrbracket^{\Phi}$$

Most cases in this proof transfer directly because the lemmas used in the λ_r case still hold here. However, the $\llbracket - \rrbracket^{-}$ outer-value preservation property does not hold directly, instead we have an extra constraint that the value not be an identifier. We don't have to deal with instantiation steps here as the term we consider in the induction is in open INF.

case $\pi_r\ a$ **:**

In the λ_r proof, this case is further decomposed by the possible reductions of $\pi_r\ a$. We have to alter the case where the projection occurs to show that a is not an identifier so that the $\llbracket - \rrbracket^{-}$ outer-value preservation result can be used. This is easily done as by assumption $(\pi_r\ a)\ \mathsf{inf}^{\circ}_{\mathsf{d}}$, and if a was an identifier, say z, then this would not hold as z would be in a destruct position.

case $a_1\ a_2$ **:**

We alter this case similarly to the last.

□

Lemma 89 (d'-d correspondence)

$$a \ \mathit{closed} \wedge\ wf[a] \ \wedge\ a \xrightarrow{l}_{d'} a' \ \wedge\ l \neq zero \implies \exists e'.\ \epsilon[a] \longrightarrow_d e' \ \wedge\ e' = \epsilon[a']$$

Proof The proof is the same as the λ_r case. The (inst-1) and (inst-2) cases follow, as they did in the λ_r case, by the inst match property. □

Lemma 90 (cd simulation) *R is a simulation from λ_c to λ_d*

Lemma 91 (d-d' correspondence)

$$a \ \mathit{closed} \wedge\ wf[a] \ \wedge\ \epsilon[a] \longrightarrow_d e' \implies \exists a', a''.\ a \xrightarrow{zeros}{}^{*}_{d'} a'' \longrightarrow_{d'} a' \ \wedge\ a'' \ \mathit{znf_d} \wedge\ e' = \epsilon[a']$$

Proof We prove the generalised statement:

$$\mathrm{wf}[a] \ \wedge\ a \ \mathsf{znf}^{\circ}_{\mathsf{d}} \ \wedge\ \epsilon[a] \longrightarrow_d e' \implies \exists a'.\ a \longrightarrow_{d'} a' \ \wedge\ e' = \epsilon[a']$$

Most cases in this proof transfer directly because the lemmas used in the λ_r case still hold here. As the instantiation rules have changed, we need to reprove the $\mathbf{let}_0\ z = a_1 \ \mathbf{in}\ a_2$, $a_1\ a_2$ and $\pi_r\ a$ cases:

case $\pi_r\ a$ **:**

In the λ_r proof this case is further decomposed by the possible reductions of the erased term. We have to add an extra case to this for the instantiation:

case $\pi_r\, \epsilon[a] = \pi_r\, (E_2.\mathbf{let}\ z = u\ \mathbf{in}\ E_2'.z)$ **:**

We can assume that a $\mathsf{znf}_\mathsf{r}^\circ$ and $\mathrm{wf}[a]$ and $\pi_r\, \epsilon[a] \longrightarrow_d \pi_r\, E_2.\mathbf{let}\ z = u\ \mathbf{in}\ E_2'.u$. We have to prove that $\pi_r\, a \longrightarrow_{d'} a'$ and $\pi_r\, (E_2.\mathbf{let}\ z = u\ \mathbf{in}\ E_2'.u) = \epsilon[a']$. By case split $\epsilon[a] = E_2.\mathbf{let}\ z = u\ \mathbf{in}\ E_2'.z$. By $\epsilon[-]$ source context (Lemma 82)for some $\hat{E}_2, \hat{a}$ we have $a = \hat{E}_2.\hat{a} \wedge \epsilon[\hat{E}_2] = E_2$, therefore $\epsilon[\hat{a}] = \mathbf{let}\ z = u\ \mathbf{in}\ E_2'.z$. By $\mathsf{znf}_\mathsf{d}^\circ$ preserved by E_3 stripping (Lemma 74) $\hat{a}$ $\mathsf{znf}_\mathsf{d}^\circ$. As $\hat{a}$ $\mathsf{znf}_\mathsf{r}^\circ$ and it erases to a let, then $\hat{a}$ must be a $\mathbf{let}_0$, as supposing that it is a $\mathbf{let}_1$ leads to a contradiction about it's ZNF property. Thus $\hat{a} = \mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2$, $\epsilon[a_1] = u$, $\epsilon[a_2] = E_2'.z$ and a_1 $\mathsf{d'val}$ by well-formedness. By $\epsilon[-]$ source context (Lemma 82)for some $\hat{E}_2{}', \hat{a}_2$ we have $a_2 = \hat{E}_2{}'.\hat{a}_2 \wedge \epsilon[\hat{E}_2{}'] = E_2'$ it follows that $\epsilon[\hat{a}_2] = z$ thus $\hat{a}_2 = z$. Moreover $a = \pi_r\, \hat{E}_2.\mathbf{let}\ z = a_1\ \mathbf{in}\ \hat{E}_2{}'.z$ which reduces under $\lambda_{d'}$ to $\pi_r\, \hat{E}_2.\mathbf{let}\ z = a_1\ \mathbf{in}\ \hat{E}_2{}'.a_1$. It is then simple enough to check that this erases to $\pi_r\, E_2.\mathbf{let}\ z = u\ \mathbf{in}\ E_2'.u$.

case $a_1\ a_2$ **:**

Similar to the above proof.

case $\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2$ **:**

We have to consider the case where this term erases to a term that can do an instantiation:

case $\mathbf{let}\ z = \epsilon[a_1]\ \mathbf{in}\ \epsilon[a_2] = \mathbf{let}\ z = u\ \mathbf{in}\ E_3.R.E_2.z$ **:**

We can assume that $\mathrm{wf}[\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2] \wedge (\mathbf{let}_0\ z = a_1\ \mathbf{in}\ a_2)$ $\mathsf{znf}_\mathsf{d}^\circ$ and $\mathbf{let}\ z = u\ \mathbf{in}\ E_3.R.E_2.z \longrightarrow_d \mathbf{let}\ z = u\ \mathbf{in}\ E_3.R.E_2.u$. By $\epsilon[-]$ source context (Lemma 82)there exists a λ_d context $\hat{E}_3$ and a $\hat{a}$ such that $a_2 = \hat{E}_3.\hat{a}$ and $\epsilon[\hat{E}_3] = E_3$, thus as erase distributes over contexts, $\epsilon[\hat{a}] = R.E_2.z$. We can see by inspection of $\epsilon[-]$ that if an erase results in an R context, then the input to erase must have been an R context, therefore let $\hat{a} = R.\hat{a}'$ for some $\hat{a}'$ then $\epsilon[R.\hat{a}'] = R.E_2.z$ and as erase distributes over contexts $\epsilon[\hat{a}'] = E_2.z$. By $\epsilon[-]$ source context (Lemma 82)there exists $\hat{E}_2$ and $\check{a}$ such that $\hat{a}' = \hat{E}_2.\check{a}$ and $\epsilon[\hat{E}_2] = E_2$ therefore $\epsilon[\check{a}] = z$ forcing $\check{a} = z$. Putting this all together $a_2 = \hat{E}_3.R.\hat{E}_2.z$, by well-formedness a_1 $\mathsf{d'val}$ and so $(\mathbf{let}\ z = a_1\ \mathbf{in}\ a_2) = (\mathbf{let}_0\ z = a_1\ \mathbf{in}\ \hat{E}_3.R.\hat{E}_2.z) \longrightarrow_{d'} \mathbf{let}_0\ z = u\ \mathbf{in}\ \hat{E}_3.R.\hat{E}_2.a_1$. More over it is easy to check that this last term erases to $\mathbf{let}\ z = u\ \mathbf{in}\ E_3.R.E_2.u$.

□

Lemma 92 (d'-c correspondence)

$$a\ \mathit{closed} \wedge \mathit{wf}[a] \wedge a \xrightarrow{l}_{d'} a' \wedge l \neq \mathit{inst} \implies \exists e'.\ [\![\, a\,]\!]^{\varnothing} \longrightarrow_c e' \wedge e' = [\![\, a'\,]\!]^{\varnothing}$$

Lemma 93 (dc simulation) *R is a weak simulation from λ_d to λ_c*

Lemma 94 (value correspondence) *if $\Phi = \Phi_k$ where*

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

and $\mathit{fv}(u) \subseteq \mathit{dom}(\Phi)$ and $\mathit{wf}[u]$ then $S[\Phi]\{\!|\ \epsilon[u]\ |\!\} = [\![\, u\,]\!]^{\Phi}$

Proof The proof follows the λ_r proof, but with an extra case necessary as variables can now be values. We give the extra case:

case z **:**

Under the assumptions $\Phi = \Phi_k;\ z \in \mathrm{dom}(\Phi)$ and $\mathrm{wf}[z]$ we are required to prove $\mathrm{S}[\Phi](z) = \Phi^*(z)$.
As $z \in \mathrm{dom}(\Phi)$, there exists $j \in 1 \,..\, k$ such that

$$\begin{array}{rcl} \mathrm{S}[\Phi](z) & = & S(\Phi_j, z \mapsto [\![\, u_{j+1} \,]\!]^{\Phi_j}(z) \\ & = & \mathrm{S}[\Phi_j][\![\, u_{j+1} \,]\!]^{\Phi_j} \end{array}$$

As $\mathrm{fv}([\![\, u_{j+1} \,]\!]^{\Phi_j}) = \varnothing$ then $\mathrm{S}[\Phi_j][\![\, u_{j+1} \,]\!]^{\Phi_j} = [\![\, u_{j+1} \,]\!]^{\Phi_j}$. To complete the proof consider $\Phi^*(z)$. As for all $v \in \mathrm{dom}(\Phi)$ it is the case that $\mathrm{fv}(v) = \varnothing$, we have that $\Phi^* = \Phi$. It follows that $\Phi^*(z) = \Phi(z) = [\![\, u_{j+1} \,]\!]^{\Phi_j}$ as required.

$\square$

Lemma 95 (typing is substitutive)

$$\Phi \vdash v{:}T \;\wedge\; z{:}T, \Phi \vdash e{:}T' \implies \Phi \vdash \{v/z\}e{:}T'$$

Lemma 96 ($\{|\ \_\ |\}$ type preservation)

$$\Phi \vdash u{:}T \implies \Phi \vdash \{|\ u\ |\}{:}T$$

Proof Follows as in the λ_r proof, but with a case for variables that is trivial. $\square$

The proof of the main theorem follows in the same way as in λ_r as the argument is purely based upon lemmas that have been reproved for the λ_d case, namely Lemma 87, Lemma 90, Lemma 80, Lemma 80, Lemma 77, Lemma 69, Lemma 71, Lemma 84, Lemma 93, Lemma 64, Lemma 76, Lemma 75, Lemma 94 and Lemma 96.

C λ_{marsh}: Sanity Properties

C.1 Unique redex/context decomposition

Theorem 97 (I.H. for Unique redex/context decomposition for λ_{marsh}) *Let e be an expression. Then (in λ_{marsh}) exactly one of the following holds:*

1. *e val: e is a value and $\neg(e\ \mathsf{var}_2)$ (value: may be grabbed or benign unbound variable).*
2. *e var_1: there exist E_3, R, E_2, z such that $E_3.R.E_2.z = e$ and $z \notin hb(E_3.R.E_2)$ (unbound variable in destruct position).*
3. *e var_2: there exist E_2, z such that $E_2.z = e$ and $z \in hb(E_2)$ (value: bound variable other than by **marshalled** Γ _).*
4. *e err_1: e err and $\neg(e\ \mathsf{var}_1)$ (fatal error).*
5. *e red: 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 (reducible).*
6. *e grb: there exist E_3', M, u such that E_3'.**marshal** $M\ u = e$ and **mark** M not around _ in E_3' (unmarked grab).*
7. *e ung: there exist E_3', M, E_2, Γ, u such that E_3'.**unmarshal** M _.E_2.**marshalled** $\Gamma\ u = e$ and **mark** M not around _ in E_3' unmarked ungrab).*

Furthermore, if such a pair, triple, quadruple, or quintuple exists then it is unique.

Proof The proof is by induction on the structure of e, and is in essence identical to the earlier proof of Theorem 13, for the destruct-time calculus. The novelty subsists entirely in the new disjuncts e grb and e ung, and in the new syntactic forms **mark** M **in** e, **marshal** $M\ e'$, **marshalled** $\Gamma\ u$, and **unmarshal** $M\ e'$. We outline below the new cases of the argument; all remaining cases simply propagate the new disjuncts unchanged, upwards through the syntax tree.

case mark M in e' :
May promote e' grb or e' ung to e red by (marshal) or (unmarshal), or may promote e' ung to e err_1 by (ungrab-err3) if rebind(fv(u), E_3) is undefined. These cases are mutually exclusive.

case marshal $M\ e'$:
If e' val or e' var_1, then e grb; otherwise, the disjunct propagates upwards.

case marshalled $\Gamma\ u$:
e val.

case unmarshal $M\ e'$:
If e' val and e' is of the form $E_2.z$, then e var_1. Otherwise, if e' val or e' var_1, then if e' is of the form E_2.**marshalled** $\Gamma\ u$, then e ung, otherwise, e err_1 by (ungrab-err1). Otherwise, the disjunct propagates upwards. □

Observe that at the top level e $\mathsf{var}_1 \implies e$ err, e $\mathsf{var}_2 \implies e$ a value, e $\mathsf{grb} \implies e$ err by (grab-err), and e $\mathsf{ung} \implies e$ err by (ungrab-err2). Hence:

Corollary 98 (Unique redex/context decomposition for λ_{marsh}) *Let e be an expression. Then (in λ_{marsh}) exactly one of the following holds:*

1. *e is a value.*

2. e err.

3. *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.*

Furthermore, if such a triple exists then it is unique.

C.2 Type preservation and partial safety

Theorem 99 (Type Preservation for λ_{marsh})
If $\vdash e{:}T$ and $e \longrightarrow e'$ then $\vdash e'{:}T$

Theorem 100 (Partial Safety for λ_{marsh})
If $\vdash e{:}T$ then $\neg(e$ err$)$.

We conjecture also that for expressions which contain only one mark, which is between top-level **let** or **letrec** declarations, and in which there is no **marshal** or **unmarshal** before that mark, then no err′ can arise.

Lemma 101 (E_2 inversion for λ_{marsh}) *If $\Gamma \vdash E_2.e{:}T$ and $dhb(E_2, dom(\Gamma))$ then $\Gamma, \Gamma(E_2) \vdash e{:}T$ and $\forall e', T'.\Gamma, \Gamma(E_2) \vdash e'{:}T' \implies \Gamma \vdash E_2.e'{:}T'$.*

Lemma 102 (E_3 inversion for λ_{marsh}) *If $\Gamma \vdash E_3.e{:}T$ and $dhb(E_3, dom(\Gamma))$ then there exists T' such that $\Gamma, \Gamma(E_3) \vdash e{:}T'$ and $\forall e'.\Gamma, \Gamma(E_3) \vdash e'{:}T' \implies \Gamma \vdash E_3.e'{:}T$.*

These change from before in having $\Gamma(E_2)$, $\Gamma(E_3)$ instead of existentially-quantified Γ'. We do not give the proofs, which are essentially as before.

Lemma 103 (bindmark(_) typing) *For all E_3, Γ, e, T, if $\Gamma, \Gamma(E_3) \vdash e{:}T$ and $\exists T', e'.\Gamma \vdash E_3.e'{:}T'$ then $\Gamma \vdash bindmark(E_3).e{:}T$.*

Proof Induction on E_3, inside out (using associativity of context composition).

Case _. Trivial.

Case $A_1.E_3$. We have $\Gamma(A_1.E_3) = \Gamma(E_3)$ and $\text{bindmark}(A_1.E_3) = \text{bindmark}(E_3)$. It remains only to note that the typing rules for each $A_1.e$ require e typable in the same type environment as $A_1.e$.

Case $A_2.E_3$.

Case (let $z_k{:}T = u$ in _)$.E_3$. Suppose $\Gamma, \Gamma((\textbf{let } z_k{:}T_0 = u \textbf{ in } \_).E_3) \vdash e{:}T$ (1) and $\Gamma \vdash (\textbf{let } z_k{:}T_0 = u \textbf{ in } \_).E_3.e'{:}T'$ (2).
By (1) and the definition of $\Gamma(\_)$ we have $\Gamma, z_k{:}T_0, \Gamma(E_3) \vdash e{:}T$ (3).
By type inversion on (2) we have $\Gamma \vdash u{:}T_0$ (4) and $\Gamma, z_k{:}T_0 \vdash E_3.e'{:}T'$ (5).
By the inductive hypothesis for (3) and (5) we have $\Gamma, z_k{:}T_0 \vdash \text{bindmark}(E_3).e{:}T$ (6).
By typing on (4) and (6) we have $\Gamma \vdash \textbf{let } z_k{:}T_0 = u \textbf{ in } \text{bindmark}(E_3).e{:}T$.
By the definition of bindmark(_) we have $\Gamma \vdash \text{bindmark}((\textbf{let } z_k{:}T_0 = u \textbf{ in } \_).E_3).e{:}T$ as required.

Case (letrec $z_k{:}T' = \lambda x_i{:}T.e_2$ in _)$.E_3$. Suppose $\Gamma, \Gamma((\textbf{letrec } z_k{:}T_0 = \lambda x_i{:}T_1.e_2 \textbf{ in } \_).E_3) \vdash e{:}T$ (1) and $\Gamma \vdash (\textbf{letrec } z_k{:}T_0 = \lambda x_i{:}T_1.e_2 \textbf{ in } \_).E_3.e'{:}T'$ (2).
W.l.g. assume also $x_i \neg \in \text{dom}(\Gamma)\Gamma((\textbf{letrec } z_k{:}T_0 = \lambda x_i{:}T_1.e_2 \textbf{ in } \_).E_3)$.
By (1) and the definition of $\Gamma(\_)$ we have $\Gamma, z_k{:}T_0, \Gamma(E_3) \vdash e{:}T$ (3).
By type inversion on (2) we have T_2 such that $T_0 = T_1 \to T_2$, $\Gamma, z_k{:}T_0, x_i{:}T_1 \vdash e_2{:}T_2$ (4) and $\Gamma, z_k{:}T_0 \vdash E_3.e'{:}T'$ (5).

By the inductive hypothesis for (3) and (5) we have $\Gamma, z_k{:}T_0 \vdash \text{bindmark}(E_3).e{:}T$ (6).
By typing on (4) and (6) we have $\Gamma \vdash \textbf{letrec}\ z_k{:}T_0 = \lambda x_i{:}T_1.e_2\ \textbf{in}\ \text{bindmark}(E_3).e{:}T$.
By the definition of bindmark(_) we have $\Gamma \vdash \text{bindmark}((\textbf{letrec}\ z_k{:}T_0 = \lambda x_i{:}T_1.e_2\ \textbf{in}\ \_).E_3).e{:}T$ as required.

Case $(\textbf{mark}\ M\ \textbf{in}\ \_).E_3$**.** Suppose $\Gamma, \Gamma((\textbf{mark}\ M\ \textbf{in}\ \_).E_3) \vdash e{:}T$ (1) and $\Gamma \vdash (\textbf{mark}\ M\ \textbf{in}\ \_).E_3.e'{:}T'$ (2).
By (1) and the definition of $\Gamma(\_)$ $\Gamma, \Gamma(E_3) \vdash e{:}T$ (3).
By type inversion on (2) $\Gamma \vdash E_3.e'{:}T'$ (4).
By the inductive hypothesis for (3) and (4) $\Gamma \vdash \text{bindmark}(E_3).e{:}T$.
By typing $\Gamma \vdash \textbf{mark}\ M\ \textbf{in}\ \text{bindmark}(E_3).e{:}T$.
By definition of bindmark(_) $\Gamma \vdash \text{bindmark}(\textbf{mark}\ M\ \textbf{in}\ \_.E_3).e{:}T$.

□

Proof (Of Theorem 99, Type Preservation for λ_{marsh}) First show that if $\Gamma \vdash e{:}T$ and $e \rightharpoonup e'$ then $\Gamma \vdash e'{:}T$. The cases here ((proj), (app), (inst-1), (inst-2), (instrec-1), (instrec-2)) are essentially identical to those for λ_d – the only differences in the reduction or typing rules are those required for the syntactical adaptations, ie with (identifier,tag) pairs instead of identifiers, and with explicit type annotations on **let** and **letrec** . The last enables us to use the simpler E_2 and E_3 inversion lemmas above.

Now prove the theorem for $\longrightarrow$.

Case (E_3 **).** By the above result and Lemma 102.

Case (marshal). Consider the reduction $E_3.\textbf{mark}\ M.E_3'.\textbf{marshal}\ M\ u \longrightarrow E_3.\textbf{mark}\ M.E_3'.\textbf{marshalled}\ (\Gamma(E_3))\ (\text{bindmark}(E_3').u)$
with dhb(E_3) and **mark** M not around _ in E_3'.
W.l.g. assume $\text{dhb}(E_3', \text{hb}(E_3))$ (this depends on the fact that $\text{hb}(\text{bindmark}(E_3')) = \text{hb}(E_3')$).
Suppose $\vdash E_3.\textbf{mark}\ M.E_3'.\textbf{marshal}\ M\ u{:}T$.
By Lemma 102 there exists T' such that $\Gamma(E_3) \vdash \textbf{mark}\ M.E_3'.\textbf{marshal}\ M\ u{:}T'$ and
$\forall e'.\Gamma(E_3) \vdash e'{:}T' \implies \vdash E_3.e'{:}T$ (*).
By inversion of typing $\Gamma(E_3) \vdash E_3'.\textbf{marshal}\ M\ u{:}T'$.
By Lemma 102 there exists T'' such that $\Gamma(E_3), \Gamma(E_3') \vdash \textbf{marshal}\ M\ u{:}T''$ and
$\forall e'.\Gamma(E_3), \Gamma(E_3') \vdash e'{:}T'' \implies \Gamma(E_3) \vdash E_3'.e'{:}T'$ (**).
By inversion of typing there exists T''' such that $T'' = \mathsf{Marsh}\ T'''$ and $\Gamma(E_3), \Gamma(E_3') \vdash u{:}T'''$.
By Lemma 103 (bindmark typing) $\Gamma(E_3) \vdash (\text{bindmark}(E_3')).u{:}T'''$.
By typing $\Gamma(E_3), \Gamma(E_3') \vdash \textbf{marshalled}\ (\Gamma(E_3))\ (\text{bindmark}(E_3').u){:}T''$.
By (**) $\Gamma(E_3) \vdash E_3'.\textbf{marshalled}\ (\Gamma(E_3))\ (\text{bindmark}(E_3').u){:}T'$.
By typing $\Gamma(E_3) \vdash \textbf{mark}\ M.E_3'.\textbf{marshalled}\ (\Gamma(E_3))\ (\text{bindmark}(E_3').u){:}T'$.
By (*) $\vdash E_3.\textbf{mark}\ M.E_3'.\textbf{marshalled}\ (\Gamma(E_3))\ (\text{bindmark}(E_3').u){:}T$.

Case (unmarshal). Consider the reduction $E_3.\textbf{mark}\ M.E_3'.\textbf{unmarshal}M.E_2.\textbf{marshalled}\ \Gamma\ u \longrightarrow E_3.\textbf{mark}\ M.E_3'.\text{S}[u]$
with dhb(E_3), dhb(E_3', hb(E_3)), S[=]rebind(Γ, thb(E_3)) defined, and **mark** M not around _ in E_3'.
Suppose $\vdash E_3.\textbf{mark}\ M.E_3'.\textbf{unmarshal}M.E_2.\textbf{marshalled}\ \Gamma\ u{:}T$.
Assume w.l.g. that $\text{dhb}(E_2, \text{hb}(E_3.\textbf{mark}\ M.E_3'))$.
Let $\Gamma' = \Gamma(E_3.\textbf{mark}\ M.E_3')$.
It is immediate from the above that $\text{dhb}(E_3.\textbf{mark}\ M.E_3', \{\})$ and $\text{dhb}(E_2, \text{hb}(E_3.\textbf{mark}\ M.E_3'))$.

By Lemma 102 there exists T' such that $\Gamma' \vdash$ **unmarshal**$M.E_2.$**marshalled** $\Gamma\ u{:}T'$ and $\forall e'.\Gamma' \vdash e'{:}T' \implies \vdash E_3.$**mark** $M.E'_3.e'{:}T$ (*)
By inversion of typing $\Gamma' \vdash E_2.$**marshalled** $\Gamma\ u{:}\mathsf{Marsh}\ T'$.
Trivially $\text{hb}(E_3.$**mark** $M.E'_3) = \text{dom}(\Gamma')$ so $\text{dhb}(E_2, \text{dom}(\Gamma'))$.
By Lemma 101 $\Gamma', \Gamma(E_2) \vdash$ **marshalled** $\Gamma\ u{:}\mathsf{Marsh}\ T'$.
By inversion of typing $\Gamma \vdash u{:}T'$.

Lemma 104 *If $rebind(\Gamma, L)$ defined then $dom(rebind(\Gamma, L)) = dom(\Gamma)$, $ran(rebind(\Gamma, L)) \subseteq \{x_i \mid \exists T.(x_i{:}T) \in L\}$, and forall $x_j \in dom(\Gamma)$, if $\Gamma(x_j) = T$ then $\exists (x_j{:}T) \in L$.*

Proof By inspection of the definition of rebind(_, _). □

Hence $\text{dom}(\text{rebind}(\Gamma, \text{thb}(E_3))) = \text{dom}(\Gamma)$, $\text{ran}(\text{rebind}(\Gamma, \text{thb}(E_3))) \subseteq \text{dom}(\Gamma(E_3))$, and forall $x_j \in \text{dom}(\Gamma)$, if $\Gamma(x_j) = T$ then $\Gamma(E_3)(\text{rebind}(\Gamma, \text{thb}(E_3))(x_j)) = T$.

Lemma 105 (variable-for-variable substitution) *If $\Gamma \vdash e{:}T$ and $S[:]dom(\Gamma) \to dom(\Gamma')$ is a variable-for-variable substitution such that $\forall x_i \in dom(\Gamma).\Gamma'(S[x_i]) = \Gamma(x_i)$ then $\Gamma' \vdash S[e]{:}T$.*

Proof Routine induction, noting that in the **marshalled** $\Gamma'\ u$ case there is nothing to do. □

Hence $\Gamma(E_3) \vdash \mathrm{S}[u]{:}T'$.
By weakening $\Gamma(E_3.$**mark** $M.E'_3) \vdash \mathrm{S}[u]{:}T'$, ie $\Gamma' \vdash \mathrm{S}[u]{:}T'$.
By (*) $\vdash E_3.$**mark** $M.E'_3.\mathrm{S}[u]{:}T$.

□

Proof (of Theorem 100, Safety for λ_{marsh})

Cases (proj-err), (app-err). These are essentially as in λ_d.

Case (ungrab-err1). Consider $E_3.$**unmarshal**$M.E_2.w$ err.

Suppose $\vdash E_3.$**unmarshal**$M.E_2.w{:}T$, $\neg\exists u, \Gamma.w =$ **marshalled** $\Gamma\ u$ (*), and $\neg\exists z_k \in \text{hb}(E_2, E_3).w = z_k$ (**).
W.l.g. $\text{dhb}(E_3, \{\})$ and $\text{dhb}(E_2, \text{hb}(E_3))$.
By Lemma 102 there exists T' such that $\Gamma(E_3) \vdash$ **unmarshal**$M.E_2.w{:}T'$.
By inversion of typing $\Gamma(E_3) \vdash E_2.w{:}\mathsf{Marsh}\ T'$.
By Lemma 101 $\Gamma(E_3), \Gamma(E_2) \vdash w{:}\mathsf{Marsh}\ T'$.
The only w forms which are typable with a grabbed type are $\mathsf{Marsh}\ \Gamma u$ and z_k. The former contradicts (*). For the latter, by (**) z_k is free in $E_3.$**unmarshal**$M.E_2.w$, which contradicts its typability in the empty context.

□

References

[AB02] Z. M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1–3):97–170, 2002.

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

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

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

[BHS+03] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Proc. ICFP 2003*, 2003.

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

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

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

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

[DE] Sophia Drossopoulou and Susan Eisenbach. Manifestations of Java dynamic linking. `http://www-dse.doc.ic.ac.uk/projects/slurp/dynamic_link/Manifest.pdf`.

[dlo] POSIX `dlopen` specification. `http://www.opengroup.org/onlinepubs/007904975/functions/dlopen.html`.

[Dug00] Dominic Duggan. Sharing in Typed Module Assembly Language. In *Proc. 3rd Workshop on Types in Compilation*, pages 85–116, 2000.

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

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

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

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

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

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

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

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

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

[Hir03] Tom Hirschowitz. *Modules mixins, modules et récursion étendue en appel par valeur*. Thèse de doctorat, Université Paris 7, 2003.

[HLW03] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Principles and Practice of Declarative Programming*, pages 160–171. ACM Press, 2003.

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

[HW00] Michael Hicks and Stephanie Weirich. A calculus for dynamic loading. Technical Report MS-CIS-00-07, University of Pennsylvania, 2000.

[HWC00] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In *Proc. 3rd Workshop on Types in Compilation*, pages 147–176, 2000.

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

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

[L$^+$01] X. Leroy et al. The Objective Caml system release 3.04 documentation, December 2001.

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

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

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

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

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

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

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

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

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

[Rou96] François Rouaix. A Web navigator with applets in Caml. In *Proc. 5th World Wide Web Conference*, pages 1365–1371, 1996.

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

[Sew97] Peter Sewell. On implementations and semantics of a concurrent programming language. In *Proceedings of CONCUR 97: Concurrency Theory (Warsaw). LNCS 1243*, pages 391–405, July 1997.

[SNC00] Albert Serra, Nacho Navarro, and Toni Cortes. DITools: Application-level support for dynamic extension and flexible composition. In *Proc. USENIX Annual Technical Conference*, pages 225–238, 2000.

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

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

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

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

This document was generated from:
Id: paper2.mng,v 1.103 2004/02/03 12:45:06 pes20 Exp
Id: common.mng,v 1.103 2004/02/03 12:13:39 gmb Exp
Id: examples.mng,v 1.10 2003/04/05 10:11:26 pes20 Exp
Id: related.mng,v 1.52 2004/02/03 12:13:39 gmb Exp