Compilation and Equivalence of Imperative Objects

Andrew D. Gordon
University of Cambridge
Computer Laboratory
adg@cl.cam.ac.uk

Paul D. Hankin
University of Cambridge
Computer Laboratory
pdh13@cam.ac.uk

S. B. Lassen
BRICS*
Department of Computer Science
University of Aarhus
thales@brics.dk

June 27, 1997

Abstract

We adopt the untyped imperative object calculus of Abadi and Cardelli as a minimal setting in which to study problems of compilation and program equivalence that arise when compiling object-oriented languages. We present both a big-step and a small-step substitution-based operational semantics for the calculus and prove them equivalent to the closure-based operational semantics given by Abadi and Cardelli. Our first result is a direct proof of the correctness of compilation to a stack-based abstract machine via a small-step decompilation algorithm. Our second result is that contextual equivalence of objects coincides with a form of Mason and Talcott's CIU equivalence; the latter provides a tractable means of establishing operational equivalences. Finally, we prove correct an algorithm, used in our prototype compiler, for statically resolving method offsets. This is the first study of correctness of an object-oriented abstract machine, and of operational equivalence for the imperative object calculus.

---

*Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

Contents

1 Motivation

The art of writing clear, precise and consistent descriptions is important across all engineering, in particular in the design of programming languages. This paper collates and extends a variety of techniques of operational semantics, a style of mathematical description widely held to be particularly suited to language descriptions. In general, mathematical descriptions of computation have long been advocated (McCarthy 1963). Mathematics is neither necessary nor sufficient to achieve clarity, precision or consistency in a language description, but it does contribute to precision and it does allow properties to be proved of a description that improve our confidence in its consistency. Operational semantics is particularly suited to language descriptions because of its mathematical simplicity, and hence its clarity.

In an operational semantics meaning is conferred on programs via a mathematical description of an interpreter for the language. Since they directly express computational behaviour, interpreters are usually easier to understand than compilers. For the same reason, operational semantics usually achieve greater clarity than semantics based entirely on translation—formalised compilation—of programs to a separate mathematical theory. At one point, operational semantics was dominated by rather low-level mathematical descriptions of interpreters, such as the SECD machine of Landin (1964). But subsequent work on applying ideas from logic and proof theory to operational semantics (Plotkin 1975; Plotkin 1981; Martin-Löf 1983; Kahn 1988) led to more abstract descriptions of interpreters that have proved successful in defining realistic programming languages, most prominently Standard ML (Milner, Tofte, and Harper 1990).

The motivation for this paper is that in spite of these attractions, the art of writing clear, precise and consistent operational semantics of programming languages remains generally poorly understood. Our response is to contribute a relatively small but accessible example of a language description based on operational semantics.

The language we describe is essentially the imperative object calculus of Abadi and Cardelli (1995a, 1995b, 1996), a small but extremely rich language that directly accommodates object-oriented, imperative and functional programming styles. Abadi and Cardelli invented the calculus to serve as a foundation for understanding object-oriented programming; in particular, they use the calculus to develop a range of increasingly sophisticated type systems for object-oriented programming. We have implemented the calculus as part of a longer term project to investigate concurrent object-oriented languages. This paper develops formal foundations and verification methods to document and better understand various aspects of our implementation.

Our system compiles the imperative object calculus to bytecodes for an abstract machine, implemented in C, based on the ZAM of Leroy (1990). We also implemented a closure-based interpreter for the calculus. A type-checker enforces the system of primitive self types of Abadi and Cardelli, but for the sake of brevity we will not discuss this type system further in this paper. Many of our semantic techniques originate in earlier studies of the λ-calculus; this paper develops these techniques further and is their first application to an object calculus.

The rest of the paper is organised as follows:

- In Section 2 we present our source language together with three forms of operational semantics. We verify their consistency with one another.

- Our target language is the instruction set of an object-oriented abstract machine, a simplification of the machine used in our implementation, and analogous to abstract machines for functional languages. Section 3 presents a formal description of our abstract machine, and a compiler from the object calculus to instructions for the abstract machine. We prove the compiler correct by adapting an idea of Rittri (1990).

- Given the formal description of our source language, we may express correctness of source-to-source transformations via operational equivalence. In Section 4, we adapt the contextual equivalence of Morris (1968), which has become the standard for studies of λ-calculi, to the imperative object calculus. We characterise it using the CIU equivalence of Mason and Talcott (1991).

- In Section 5, we exercise operational equivalence by specifying and verifying a simple optimisation that resolves at compile-time certain method labels to integer offsets.

We discuss related work at the ends of Sections 2, 3, 4 and 5. Finally, we review the contributions of the paper in Section 6.

Anyone desiring to experiment with our implementation is asked to contact the authors.

2 An Imperative Object Calculus

In this section, we present the syntax of an imperative object calculus, together with three forms of operational semantics, which we prove to be consistent with one another.

2.1 Syntax of the Calculus

We begin with the syntax of an untyped imperative object calculus, the **impς** calculus of Abadi and Cardelli (1996) augmented to include store locations as terms. Let x, y, and z range over an infinite collection of *variables*. Let ι range over an infinite collection of *locations*, the addresses of objects in the store.

The set of *terms* of the calculus is given as follows:

$a, b ::=$	term
x	variable
ι	location
$[\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$	object (ℓ_i distinct)
$a.\ell$	method selection
$a.\ell \Leftarrow \varsigma(x)b$	method update
$clone(a)$	cloning
$let\ x = a\ in\ b$	let

Informally, when an object is created, it is put at a fresh location, ι, in the store, and referenced thereafter by ι. Method selection runs the body of the method with the self parameter (the x in $\varsigma(x)b$) bound to the location of the object containing the method. Method update allows objects in the store to be dynamically altered. Cloning makes a fresh copy of an object in the store at a new location. The reader unfamiliar with object calculi is encouraged to consult the book of Abadi and Cardelli (1996) for many examples and a discussion of the design choices that led to this calculus.

Here are the scoping rules for variables: in a method $\varsigma(x)b$, variable x is bound in b; in $let\ x = a\ in\ b$, variable x is bound in b. If ϕ is a phrase of syntax we write $fv(\phi)$ for the set of variables that occur free in ϕ. We say phrase ϕ is *closed* if $fv(\phi) = \varnothing$. We write $\phi\{\!\{\psi/x\}\!\}$ for the substitution of phrase ψ for each free occurrence of variable x in phrase ϕ. We identify all phrases of syntax up to alpha-conversion; hence $a = b$, for instance, means that we can obtain term b from term a by systematic renaming of bound variables. Let o range over objects, terms of the form $[\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$. In general, the notation $\phi_i \ ^{i \in 1..n}$ means $\phi_1, \ldots, \phi_n$.

Unlike Abadi and Cardelli, we include locations in the syntax of terms and we do not identify objects up to re-ordering of methods since the order of methods in an object is important for an algorithm we present in Section 5 for statically resolving method offsets.

We include locations in terms in order to express the dynamic behaviour of the calculus using a substitution-based operational semantics. If ϕ is a phrase of syntax, let $locs(\phi)$ be the set of locations that occur in ϕ. Let a term a be a *static term* if $locs(a) = \varnothing$. The static terms correspond to the source syntax accepted by our compiler.

First, here is how to express updateable references using objects with a single *ref* method:

$$\begin{aligned} ref(a) &\stackrel{\text{def}}{=} let\ x = a\ in\ [ref = \varsigma(y)x] \\ a := b &\stackrel{\text{def}}{=} let\ x = b\ in\ a.ref \Leftarrow \varsigma(y)x \\ !a &\stackrel{\text{def}}{=} a.ref \end{aligned}$$

As a second example, here is an encoding of the call-by-value λ-calculus, like an encoding due to Abadi and Cardelli but with right-to-left evaluation of function application:

$$\begin{aligned} \lambda(x)b &\stackrel{\text{def}}{=} [arg = \varsigma(z)z.arg, val = \varsigma(s)let\ x = s.arg\ in\ b] \\ b(a) &\stackrel{\text{def}}{=} let\ y = a\ in\ (b.arg \Leftarrow \varsigma(z)y).val \end{aligned}$$

where $y \neq z$, and s and y do not occur free in b. Given updateable methods, we can easily extend this encoding to express an ML-style call-by-value λ-calculus with updateable references.

Unlike Abadi and Cardelli's imperative λ-calculus, the **imp**λ calculus, this encoding does not permit assignments to bound variables.

In order to avoid the overhead of encoding, our implementation treats functions and function application as language primitives and compiles them to dedicated abstract machine codes based on the ZAM of Leroy (1990). In this implementation, a function application $b(a)$ is evaluated right-to-left on grounds of efficiency.

Although functions are derivable, for the purpose of the operational semantics of this section and the abstract machine and compiler in the next, Section 3, we consider an extended calculus that includes functions and function application. This is partly because a realistic implementation would include functions (procedures) as primitive, and partly to demonstrate the applicability of the techniques of these sections to a λ-calculus with state. We do not use this extended calculus in Section 4 or in Section 5. The techniques used in the study of operational equivalence in Section 4 are well understood for λ-calculi with state. The optimisation of method access in Section 5 is independent of the presence of primitive functions.

The syntax of the extended calculus is given by:

$a, b ::=$	terms

$\ldots$	as previously
$\lambda(x)b$	function
$b(a)$	application

In a function $\lambda(x)b$, variable x is bound in b. Unlike Abadi and Cardelli's imperative λ-calculus, the **imp**λ calculus, our extended calculus does not permit assignments to bound variables.

Throughout this paper, and in our implementation, we adopt the convention that a function application $b(a)$ is evaluated right-to-left; a is evaluated before b. In making this choice we are following Leroy (1990), who proposes it on grounds of efficiency. Adopting a left-to-right evaluation order would have little effect on the contents of this paper, but would adversely affect the performance of our implementation.

We finish this section by fixing notation for finite lists and finite maps. We write finite lists in the form $[\phi_1, \ldots, \phi_n]$, which we usually write as $[\phi_i{}^{i \in 1..n}]$. Let $\psi :: [\phi_i{}^{i \in 1..n}] = [\psi, \phi_i{}^{i \in 1..n}]$. Let $[\phi_i{}^{i \in 1..m}]@[\psi_j{}^{j \in 1..n}] = [\phi_i{}^{i \in 1..m}, \psi_j{}^{j \in 1..n}]$.

Let a *finite map*, f, be a list of the form $[x_i \mapsto \phi_i{}^{i \in 1..n}]$, where the x_i are distinct. When $f = [x_i \mapsto \phi_i{}^{i \in 1..n}]$ is a finite map, let $dom(f) = \{x_i{}^{i \in 1..n}\}$. When finite map $f = f'@[x \mapsto \phi]@f''$, let $f(x) = \phi$. When f and g are finite maps, let the map $f + (x \mapsto \phi)$, be $f'@[x \mapsto \phi]@f''$ if $f = f'@[x \mapsto \psi]@f''$, otherwise $(x \mapsto \phi) :: f$.

2.2 Small-Step Substitution-Based Semantics

The goal of this section is to specify a relation, $c \to d$, where c and d are each *configurations* consisting of a closed term paired with an object store. Intuitively, $c \to d$ means that the program state represented by c takes a single computation step to reach d. We present this operational semantics using reduction contexts introduced in the study of imperative λ-calculi by Felleisen and Friedman (1986). We say this is a small-step semantics because it defines individual steps of computation. We say it is substitution-based because it is defined in terms of the substitution primitive, $-\{\!\{v/x\}\!\}$, that substitutes values for variables. We use this semantics in Section 3 to prove correctness of compilation. In the course of this paper, we use the symbol $\to$ for several small-step relations; we often refer to such relations as *reduction* relations.

Let a *store*, σ, be a finite map from locations to objects. Each stored object consists of a collection of labelled methods. The methods may be updated individually. Abadi and Cardelli use a method store, a finite map from locations to methods, in their operational semantics of imperative objects.

We prefer to use an object store, as it explicitly represents the grouping of methods in objects. We discuss the connection between our semantics and that of Abadi and Cardelli in Section 4.6.

$\sigma ::= [\iota_i \mapsto o_i \ ^{i \in 1..n}]$	object store (ι_i distinct)
$c, d ::= (a, \sigma)$	configuration

We write $\vdash \sigma\ ok$, to mean that a store σ is well formed, if and only if $fv(\sigma(\iota)) = \varnothing$ and $locs(\sigma(\iota)) \subseteq dom(\sigma)$ for each $\iota \in dom(\sigma)$. We write $\vdash (a, \sigma)\ ok$, to mean that a configuration (a, σ) is well formed, if and only if $fv(a) = \varnothing$, $locs(a) \subseteq dom(\sigma)$ and $\vdash \sigma\ ok$.

To define the reduction relation we need the syntactic concepts of *values* and *reduction contexts*. A value is either a location or a function. A reduction context, $\mathcal{R}$, is a term given by the following grammar, with one free occurrence of a distinguished variable, $\bullet$, which represents 'the point of execution' in $\mathcal{R}$:

$u, v ::= \iota \mid \lambda(x)b$	value
$\mathcal{R} ::= \bullet \mid \mathcal{R}.\ell \mid \mathcal{R}.\ell \Leftarrow \varsigma(x)b$	reduction context
$\quad \mid clone(\mathcal{R}) \mid let\ x = \mathcal{R}\ in\ b$	
$\quad \mid \mathcal{R}(v) \mid a(\mathcal{R})$	

Since there is exactly one free occurrence of $\bullet$ in any reduction context, if $\mathcal{R}.\ell \Leftarrow \varsigma(x)b$ is a reduction context, $\bullet \notin fv(b) - \{x\}$. For the same reason, if $let\ x = \mathcal{R}\ in\ b$, $\mathcal{R}(a)$ and $v(\mathcal{R})$ are reduction contexts, $\bullet \notin fv(b) - \{x\}$, $\bullet \notin fv(a)$ and $\bullet \notin fv(v)$, respectively. We write $\mathcal{R}[a]$ for the outcome of substituting term a (not necessarily a value) for the single occurrence of the hole $\bullet$ in a reduction context $\mathcal{R}$. No variables are ever captured by this operation, since the hole in a reduction context does not appear in the scope of any bound variables.

Let the small-step substitution-based reduction relation, $c \to d$, be given by the following axiom schemes:

(Red Object)
$$\frac{\sigma' = (\iota \mapsto o) :: \sigma \quad \iota \notin dom(\sigma)}{(\mathcal{R}[o], \sigma) \to (\mathcal{R}[\iota], \sigma')}$$

(Red Select)
$$\frac{\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}] \quad j \in 1..n}{(\mathcal{R}[\iota.\ell_j], \sigma) \to (\mathcal{R}[b_j \{\!\{ \iota / x_j \}\!\}], \sigma)}$$

(Red Update)
$$\frac{\begin{array}{l}\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..n}] \quad j \in 1..n \\ \sigma' = \sigma + (\iota \mapsto [\ell_i = \varsigma(x_i)b_i \ ^{i \in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i \ ^{i \in j+1..n}])\end{array}}{(\mathcal{R}[\iota.\ell_j \Leftarrow \varsigma(x)b], \sigma) \to (\mathcal{R}[\iota], \sigma')}$$

(Red Clone)

$$\frac{\sigma(\iota) = o \quad \sigma' = (\iota' \mapsto o) :: \sigma \quad \iota' \notin dom(\sigma)}{(\mathcal{R}[clone(\iota)], \sigma) \to (\mathcal{R}[\iota'], \sigma')}$$

(Red Let)

$$\frac{}{(\mathcal{R}[let\ x = v\ in\ b], \sigma) \to (\mathcal{R}[b\{\!\{v/x\}\!\}], \sigma)}$$

(Red Appl)

$$\frac{}{(\mathcal{R}[(\lambda(x)b)(v)], \sigma) \to (\mathcal{R}[b\{\!\{v/x\}\!\}], \sigma)}$$

The outcome of reducing a well formed configuration is itself a well formed configuration. Moreover, reduction may increase, but not decrease, the domain of the store of a configuration:

Lemma 2.1 *Suppose* $\vdash (a, \sigma)\ ok$ *and* $(a, \sigma) \to (a', \sigma')$. *Then* $\vdash (a', \sigma')\ ok$ *and* $dom(\sigma) \subseteq dom(\sigma')$.

Proof By inspection of the reduction rules. □

Let a configuration c be *terminal* if and only if there is a store σ and a value v such that $c = (v, \sigma)$. We say that a configuration c *converges*, $c\!\Downarrow$, if and only if there is a terminal configuration d such that $c \to^* d$. We say that a configuration c *diverges* if and only if there is an infinite sequence of configurations c_1, c_2, ... such that $c \to c_1 \to c_2 \to \cdots$.

For instance, consider the configuration:

$$(let\ x = [\ell = \varsigma(s)s.\ell]\ in\ clone(x).\ell \Leftarrow \varsigma(s)x, [])$$

This is not terminal, but it converges because of the following reduction sequence:

$$\begin{array}{ll} & (let\ x = [\ell = \varsigma(s)s.\ell]\ in\ clone(x).\ell \Leftarrow \varsigma(s)x, []) \\ \to & (let\ x = \iota_1\ in\ clone(x).\ell \Leftarrow \varsigma(s)x, [\iota_1 \mapsto [\ell = \varsigma(s)s.\ell]]) \\ \to & (clone(\iota_1).\ell \Leftarrow \varsigma(s)\iota_1, [\iota_1 \mapsto [\ell = \varsigma(s)s.\ell]]) \\ \to & (\iota_2.\ell \Leftarrow \varsigma(s)\iota_1, [\iota_2 \mapsto [\ell = \varsigma(s)s.\ell]], \iota_1 \mapsto [\ell = \varsigma(s)s.\ell]) \\ \to & (\iota_2, [\iota_2 \mapsto [\ell = \varsigma(s)\iota_1]], \iota_1 \mapsto [\ell = \varsigma(s)s.\ell]) \end{array}$$

(This reduction sequence illustrates that fresh locations are allocated at the beginning of the store.)

Consider now the following configuration:

$$([\ell = \varsigma(s)s.\ell].\ell, [])$$

It diverges because of the following reduction sequence:

$$\begin{array}{rcl} ([\ell = \varsigma(s)s.\ell].\ell, []) & \to & (\iota.\ell, [\iota \mapsto [\ell = \varsigma(s)s.\ell]]) \\ & \to & (\iota.\ell, [\iota \mapsto [\ell = \varsigma(s)s.\ell]]) \\ & \to & \cdots \end{array}$$

Next we show that reduction, $\to$, is deterministic up to the choice of freshly allocated locations in rules (Red Object) and (Red Clone). To state this precisely, we need a couple of definitions. First, we define a predicate which asserts that the domain of the store of a configuration includes a set w of locations: let the predicate $w \vdash (a, \sigma)$ hold if and only if $\vdash (a, \sigma)\ ok$ and $w \subseteq dom(\sigma)$. Second, we define *structural equivalence at* w, $\equiv_w$, for any finite set w of locations, as the least relation on configurations closed under the following rules.

(Struct Refl) (Struct Trans)

$$\frac{w \vdash c}{c \equiv_w c} \qquad \frac{c \equiv_w c' \quad c' \equiv_w c''}{c \equiv_w c''}$$

(Struct Rename)

$$\frac{w \vdash (a, \sigma) \quad \iota \in dom(\sigma) - w \quad \iota' \notin dom(\sigma)}{(a, \sigma) \equiv_w (a\{\!\{\iota'/\iota\}\!\}, \sigma\{\!\{\iota'/\iota\}\!\})}$$

In this definition the notation $a\{\!\{\iota'/\iota\}\!\}$ denotes the outcome of replacing every occurrence of location ι in a by ι'; and $\sigma\{\!\{\iota'/\iota\}\!\}$ denotes the outcome of renaming location ι of store σ to ι', and applying this substitution to each of the objects in the store. An easy induction establishes that $c \equiv_w d$ implies that $w \vdash c$ and $w \vdash d$. Roughly, $c \equiv_w d$ means that the locations in w are all included in the domains of the stores of both c and d, and that c may be obtained from d by a series of renamings of the locations outside w.

Lemma 2.2 *Relation* $\equiv_w$ *is symmetric, and hence is an equivalence relation.*

Proof Suppose $c \equiv_w c'$, then $c' \equiv_w c$ follows by an induction on the derivation of $c \equiv_w c'$. Cases (Struct Refl) and (Struct Trans) are easy. In the case of (Struct Rename), we must show $(a\{\!\{\iota'/\iota\}\!\}, \sigma\{\!\{\iota'/\iota\}\!\}) \equiv_w (a, \sigma)$ when $(a, \sigma) \equiv_w (a\{\!\{\iota'/\iota\}\!\}, \sigma\{\!\{\iota'/\iota\}\!\})$ derives from $w \vdash (a, \sigma)$, $\iota \in dom(\sigma) - w$ and

$\iota' \notin dom(\sigma)$. From $w \vdash (a,\sigma)$ it follows that $locs(a) \cup locs(\sigma) \cup w \subseteq dom(\sigma)$. Therefore $\iota' \notin locs(a) \cup locs(\sigma)$. Hence we have:

$$\begin{aligned} a\{\!\!\{\iota'/\iota\}\!\!\}\{\!\!\{\iota/\iota'\}\!\!\} &= a && (1)\\ \sigma\{\!\!\{\iota'/\iota\}\!\!\}\{\!\!\{\iota/\iota'\}\!\!\} &= \sigma && (2)\end{aligned}$$

From $(a,\sigma) \equiv_w (a\{\!\!\{\iota'/\iota\}\!\!\}, \sigma\{\!\!\{\iota'/\iota\}\!\!\})$ it follows that $w \vdash (a\{\!\!\{\iota'/\iota\}\!\!\}, \sigma\{\!\!\{\iota'/\iota\}\!\!\})$. We have $\iota' \notin dom(\sigma)$ and $w \subseteq dom(\sigma)$, and $\iota \in dom(\sigma) - w$, that is, $\iota \in dom(\sigma)$ but $\iota \notin w$. Therefore $\iota' \in dom(\sigma\{\!\!\{\iota'/\iota\}\!\!\})$ but $\iota' \notin w$, that is, $\iota' \in dom(\sigma\{\!\!\{\iota'/\iota\}\!\!\}) - w$. Moreover $\iota \notin dom(\sigma\{\!\!\{\iota'/\iota\}\!\!\})$, since we may conclude that $\iota \neq \iota'$ from $\iota \in dom(\sigma)$ but $\iota' \notin dom(\sigma)$. By (Struct Rename), $w \vdash (a\{\!\!\{\iota'/\iota\}\!\!\}, \sigma\{\!\!\{\iota'/\iota\}\!\!\})$, $\iota' \in dom(\sigma\{\!\!\{\iota'/\iota\}\!\!\}) - w$ and $\iota \notin dom(\sigma\{\!\!\{\iota'/\iota\}\!\!\})$ together imply

$$\begin{aligned} (a\{\!\!\{\iota'/\iota\}\!\!\}, \sigma\{\!\!\{\iota'/\iota\}\!\!\}) &\equiv_w (a\{\!\!\{\iota'/\iota\}\!\!\}\{\!\!\{\iota/\iota'\}\!\!\}, \sigma\{\!\!\{\iota'/\iota\}\!\!\}\{\!\!\{\iota/\iota'\}\!\!\})\\ &= (a,\sigma)\end{aligned}$$

the desired equation, where the second step appeals to equations (1) and (2). $\square$

The $\to$ relation is deterministic up to structural equivalence:

Proposition 2.3 *Suppose $w \vdash c$. Then $c \to c'$ and $c \to c''$ imply $c' \equiv_w c''$.*

Proof By case analysis of the derivation of $c \to c'$. Here is one case:

(Red Object) Here $c = (\mathcal{R}[o], \sigma)$ and $c' = (\mathcal{R}[\iota'], \sigma')$ where $\sigma' = (\iota' \mapsto o) :: \sigma$ and $\iota' \notin dom(\sigma)$. Since $w \vdash c$, c is well formed and therefore $\iota' \notin locs(\mathcal{R})$. Only (Red Object) may derive $c \to c''$, so $c'' = (\mathcal{R}[\iota''], \sigma'')$ where $\sigma'' = (\iota'' \mapsto o)::\sigma$ and $\iota'' \notin dom(\sigma)$. If $\iota' = \iota''$, $c' \equiv_w c''$ by (Struct Refl). Otherwise, $\iota' \neq \iota''$, so $\iota'' \notin dom(\sigma')$. Since $w \subseteq dom(\sigma)$ and $\iota' \notin dom(\sigma)$, $\iota' \in dom(\sigma') - w$. By (Struct Rename), using $\iota' \notin locs(\mathcal{R})$,

$$(\mathcal{R}[\iota'], \sigma') \equiv_w (\mathcal{R}[\iota']\{\!\!\{\iota''/\iota'\}\!\!\}, \sigma'\{\!\!\{\iota''/\iota'\}\!\!\}) = (\mathcal{R}[\iota''], \sigma''),$$

that is, $c' \equiv_w c''$.

The case for (Red Clone) is similar. If $c \to c'$ was derived using any of the other rules, and $c \to c''$, then in fact $c' = c''$; hence $c' \equiv_w c''$. $\square$

In particular, whenever (a,σ) is well formed and $(a,\sigma) \to^* d$, the configuration d is unique up to structural equivalence at $dom(\sigma)$, that is, up to the renaming of any newly generated locations in the store component of d.

Let a configuration c be *stuck* if and only if c is not terminal, but there is no d with $c \to d$. Examples are $(\iota.\ell, [\iota \mapsto []])$ and $(\iota.\ell, [])$. We say that a configuration, c, *goes wrong* if and only if there is a stuck configuration, d, such that $c \to^* d$.

Configurations related by structural equivalence at w possess the following properties:

Lemma 2.4 *Suppose $c \equiv_w c'$.*

(1) *c is terminal implies c' is terminal.*

(2) *c is stuck implies c' is stuck.*

(3) *$c \to d$ implies there exists d' such that $c' \to d'$ and $d \equiv_w d'$.*

Proof Parts (1) and (3) follow by inductions on the derivation of $c \equiv_w c'$. Then (2) follows from (1), (3) and the symmetry of $\equiv_w$, Lemma 2.2. $\square$

The symmetry of structural equivalence at w implies that, whenever $c \equiv_w c'$, then (1) c converges just if c' converges, (2) c goes wrong just if c' goes wrong, and (3) c diverges just if c' diverges.

Proposition 2.5 *For any well formed configuration c, exactly one of the following holds:*

(1) *c converges,*

(2) *c goes wrong,*

(3) *c diverges.*

Proof If there is no computation $c \to^* d$ to a terminal or stuck configuration d, then every reduction sequence from c is infinite (or extends to an infinite sequence), so (3) holds and (1) and (2) are false.

Otherwise, there is a least n such that $c \to^n d$, for some terminal or stuck configuration d. Suppose d is terminal—the case when d is stuck is analogous. Then (1) holds. By induction on n we prove that (2) and (3) are false. If $n = 0$, (2) and (3) are false because a terminal configuration is not stuck and because there is no reduction $d \to d'$ from a terminal configuration. If $n > 0$, there is c' such that $c \to c'$ and $c' \to^{n-1} d$. By induction hypothesis, c' does not go wrong and does not diverge. For any other reduction $c \to c''$, we have $c' \equiv_\varnothing c''$, by Proposition 2.3. As a consequence of Lemma 2.4, if c'' goes wrong or diverges, so does c'. Therefore there is no reduction $c \to c''$ such that c'' goes wrong or diverges. Since c is not stuck, we get that c cannot go wrong or diverge, that is, (2) and (3) are false, as required. $\square$

Finally, we later need the following property of reduction contexts, which may easily be proved.

Lemma 2.6 *For every closed reduction context $\mathcal{R}$ with $locs(\mathcal{R}) \subseteq dom(\sigma)$,*

(1) $(a, \sigma) \to (a', \sigma')$ *implies* $(\mathcal{R}[a], \sigma) \to (\mathcal{R}[a'], \sigma')$, *and*

(2) $(\mathcal{R}[a], \sigma) \to^* (v, \sigma')$ *implies there is a configuration* (v', σ'') *such that* $(a, \sigma) \to^* (v', \sigma'')$ *and* $(\mathcal{R}[v'], \sigma'') \to^* (v, \sigma')$.

Part (2) implies that if (a, σ) goes wrong or diverges, so does $(\mathcal{R}[a], \sigma)$.

2.3 Big-Step Substitution-Based Semantics

In this section, we specify a relation, $c \Downarrow d$, where again c and d are configurations, but this time with the intuition that d is the final outcome of many computation steps starting from c. We say this is a big-step semantics because it relates a configuration to the final outcome of taking many individual steps of computation. It is defined in terms of the substitution primitive, $-\{\!\!\{v/x\}\!\!\}$, like the small-step relation, $\to$, of the previous section. Unlike the $\to$ relation, the $\Downarrow$ relation is defined inductively. We exploit its induction principle in the proof of Proposition 5.3, the crux of Section 5. In the course of this paper, we use the symbol $\Downarrow$ for several big-step relations; we often refer to such relations as *evaluation* relations.

Let the big-step substitution-based evaluation relation, $c \Downarrow d$, be the relation on configurations inductively defined by the following rules:

(Subst Value)

$$\frac{}{(v, \sigma) \Downarrow (v, \sigma)}$$

(Subst Object)

$$\frac{\sigma_1 = (\iota \mapsto o) :: \sigma_0 \quad \iota \notin dom(\sigma_0)}{(o, \sigma_0) \Downarrow (\iota, \sigma_0)}$$

(Subst Select) (where $j \in 1..n$)

$$\frac{(a, \sigma_0) \Downarrow (\iota, \sigma_1) \quad \sigma_1(\iota) = [\ell_i = \varsigma(x_i)b_i \; ^{i \in 1..n}] \quad (b_j\{\!\!\{\iota/x_j\}\!\!\}, \sigma_1) \Downarrow (v, \sigma_2)}{(a.\ell_j, \sigma_0) \Downarrow (v, \sigma_2)}$$

(Subst Update) (where $j \in 1..n$)

$$\frac{\begin{array}{l}(a, \sigma_0) \Downarrow (\iota, \sigma_1) \quad \sigma_1(\iota) = [\ell_i = \varsigma(x_i)b_i \; ^{i \in 1..n}] \\ \sigma_2 = \sigma_1 + (\iota \mapsto [\ell_i = \varsigma(x_i)b_i \; ^{i \in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i \; ^{i \in j+1..n}])\end{array}}{(a.\ell_j \Leftarrow \varsigma(x)b, \sigma_0) \Downarrow (\iota, \sigma_2)}$$

(Subst Clone)

$$\frac{(a, \sigma_0) \Downarrow (\iota, \sigma_1) \quad \sigma_1(\iota) = o \quad \sigma_2 = (\iota' \mapsto o) :: \sigma_1 \quad \iota' \notin dom(\sigma_1)}{(clone(a), \sigma_0) \Downarrow (\iota', \sigma_2)}$$

(Subst Let)

$$\frac{(a, \sigma_0) \Downarrow (v, \sigma_1) \quad (b\{\!\{v/x\}\!\}, \sigma_1) \Downarrow (u, \sigma_2)}{(let\ x = a\ in\ b, \sigma_0) \Downarrow (u, \sigma_2)}$$

(Subst Appl)

$$\frac{(a, \sigma_0) \Downarrow (u, \sigma_1) \quad (b, \sigma_1) \Downarrow (\lambda(x)b', \sigma_2) \quad (b'\{\!\{u/x\}\!\}, \sigma_2) \Downarrow (v, \sigma_3)}{(b(a), \sigma_0) \Downarrow (v, \sigma_3)}$$

The big-step and small-step substitution semantics are consistent with one another in the following sense:

Theorem 2.7

(1) *Whenever* $c \Downarrow d$*,* d *is terminal and* $c \rightarrow^* d$*.*

(2) *Whenever* $c \rightarrow^* d$ *and* d *is terminal,* $c \Downarrow d$*.*

Proof

(1) By induction on the derivation of $c \Downarrow d$. The details are routine.

(2) One can prove by induction on n that $c \Downarrow d$ whenever $c \rightarrow^n d$ and d is terminal. Again, the details are routine. □

To see why we require d to be terminal in part (2), consider the stuck configuration $c = ([].\ell, [])$. We have $c \rightarrow^* c$, but there is no d such that $c \Downarrow d$.

The big-step relation, $\Downarrow$, is deterministic in the following sense:

Proposition 2.8 *Whenever* $w \vdash c$*,* $c \Downarrow c'$ *and* $c \Downarrow c''$ *imply* $c' \equiv_w c''$*.*

Proof Suppose that $c \Downarrow c'$ and $c \Downarrow c''$. By Theorem 2.7(1), both c' and c'' are terminal and there are m and n such that $c \rightarrow^m c'$ and $c \rightarrow^n c''$. Without loss of generality, suppose that $m \leq n$. There must be d such that $c \rightarrow^m d$ and $d \rightarrow^{n-m} c''$. By Proposition 2.3 and Lemma 2.4(3), $c' \equiv_w d$. It follows, by Lemma 2.4, that d is terminal, and therefore that $c'' = d$. Hence we have that $c' \equiv_w c''$. □

2.4 Big-Step Closure-Based Semantics

In this section we present an operational semantics for the imperative object calculus, based on the one in Chapter 10 of Abadi and Cardelli (1996) but with the addition of functions. It is in the same style as the dynamic semantics of expressions in the definition of Standard ML (Milner, Tofte, and

Harper 1990). Unlike the semantics of the previous sections, it uses closures, rather than a substitution primitive, to link variables to their values. Like the semantics of the previous section, it is a big-step semantics, an evaluation relation, denoted by $\Downarrow$. The main result of this section is a proof of consistency between the closure-based semantics and the substitution-based semantics of the previous section.

$U, V ::=$	closure-based value
$\quad \iota$	$\quad$ location
$\quad (S, \lambda(x)b)$	$\quad$ function closure
$S ::= [x_i \mapsto V_i\ ^{i \in 1..n}]$	stack (x_i distinct)
$O ::= [\ell_i = (S_i, \varsigma(x_i)b_i)\ ^{i \in 1..n}]$	object value
$\Sigma ::= [\iota_i \mapsto O_i\ ^{i \in 1..n}]$	store
$C, D ::=$	configuration
$\quad ((S, a), \Sigma)$	$\quad$ initial configuration
$\quad (V, \Sigma)$	$\quad$ terminal configuration

A stack $S = [x_i \mapsto V_i\ ^{i \in 1..n}]$ is a finite map that binds variables to their values. A value is either a location, ι, or a closure of the form $(S, \lambda(x)b)$ where the stack S maps each variable free in b to a value. A store Σ is a finite map sending locations to object values, which are of the form $O = [\ell_i = (S_i, \varsigma(x_i)b_i)\ ^{i \in 1..n}]$, where for each i, stack S_i maps each variable free in the method $\varsigma(x_i)b_i$ to its value. An initial configuration consists of a closure (S, a), together with a store Σ that maps locations occurring in (S, a) to object values. A terminal configuration is simply a value paired with a store. Note that a configuration of the form (V, Σ) where $V = (S, \lambda(x)b)$ is both initial and terminal.

We use uppercase metavariables for the entities used in our closure-based semantics; they mostly correspond to lowercase metavariables ranging over corresponding entities used in the substitution-based semantics. For example, σ is a store used in the two substitution-based semantics, and Σ is a store used in the closure-based semantics. We refer to both entities as stores, relying on the case of the metavariable to indicate which kind of store is meant.

Let the big-step substitution-based evaluation relation, $C \Downarrow D$, be the relation on configurations inductively defined by the following rules:

(Closure x)

$$\frac{S(x) = V}{((S, x), \Sigma) \Downarrow (V, \Sigma)}$$

(Closure Value)

$$\frac{}{((S, \lambda(x)b), \Sigma) \Downarrow ((S, \lambda(x)b), \Sigma)}$$

(Closure Select)

$$\frac{((S,a),\Sigma_0) \Downarrow (\iota,\Sigma_1) \quad \Sigma_1(\iota) = [\ell_i = (S_i, \varsigma(x_j)b_i)^{\,i\in 1..n}] \\ j \in 1..n \quad x_j \notin dom(S_j) \quad (((x_j \mapsto \iota) :: S_j, b_j),\Sigma_1) \Downarrow (V,\Sigma_2)}{((S,a.\ell_j),\Sigma_0) \Downarrow (V,\Sigma_2)}$$

(Closure Update)

$$\frac{((S,a),\Sigma_0) \Downarrow (\iota,\Sigma_1) \quad \Sigma_1(\iota) = [\ell_i = (S_i, \varsigma(x_i)b_i)^{\,i\in 1..n}] \quad j \in 1..n \\ O = [\ell_i = (S_i, \varsigma(x_i)b_i)^{\,i\in 1..j-1}, \ell_j = (S, \varsigma(x)b), \ell_i = (S_i, \varsigma(x_i)b_i)^{\,i\in j+1..n}]}{((S, a.\ell_j \Leftarrow \varsigma(x)b),\Sigma_0) \Downarrow (\iota, (\iota \mapsto O) + \Sigma_1)}$$

(Closure Object) (where $x_i \notin dom(S)$ for each $i \in 1..n$)

$$\frac{\Sigma_1 = (\iota \mapsto [\ell_i = (S, \varsigma(x_i)b_i)^{\,i\in 1..n}]) :: \Sigma_0 \quad \iota \notin dom(\Sigma_0)}{((S, [\ell_i = \varsigma(x_i)b_i^{\,i\in 1..n}]),\Sigma_0) \Downarrow (\iota,\Sigma_1)}$$

(Closure Clone)

$$\frac{((S,a),\Sigma_0) \Downarrow (\iota,\Sigma_1) \quad \Sigma_1(\iota) = O \quad \Sigma_2 = (\iota' \mapsto O) :: \Sigma_1 \quad \iota' \notin dom(\Sigma_1)}{((S, clone(a)),\Sigma_0) \Downarrow (\iota',\Sigma_2)}$$

(Closure Let)

$$\frac{((S,a),\Sigma_0) \Downarrow (V,\Sigma_1) \quad x \notin dom(S) \quad (((x \mapsto V) :: S, b),\Sigma_1) \Downarrow (U,\Sigma_2)}{((S, let\ x = a\ in\ b),\Sigma_0) \Downarrow (U,\Sigma_2)}$$

(Closure Appl)

$$\frac{((S,a),\Sigma_0) \Downarrow (U,\Sigma_1) \quad ((S,b),\Sigma_1) \Downarrow ((S', \lambda(x)b'),\Sigma_2) \quad x \notin dom(S') \\ (((x \mapsto U) :: S', b'),\Sigma_2) \Downarrow (V,\Sigma_3)}{((S, b(a)),\Sigma_0) \Downarrow (V,\Sigma_3)}$$

These rules are almost identical to the ones from Chapter 10 of Abadi and Cardelli (1996), except for the inclusion of functions and except that locations contain objects in our semantics but methods in theirs, as discussed earlier (and in Section 4.6).

The semantics does indeed relate initial and terminal configurations:

Lemma 2.9 *Whenever $C \Downarrow D$, C is an initial configuration and D is a terminal configuration.*

Proof By induction on the derivation of $C \Downarrow D$. □

To establish a correspondence between this closure-based semantics and the substitution-based semantics of Section 2.3, we introduce several relations that unload the entities used in the closure-based semantics by turning closures into substitutions. Let s range over a substitution of the form $[v_i/x_i{}^{i\in 1..n}]$

where the x_i are distinct and each v_i is closed. We use the symbol $\leadsto$ for each of five unloading relations.

$V \leadsto v$	value unloading
$S \leadsto s$	stack unloading
$O \leadsto o$	object unloading
$\Sigma \leadsto \sigma$	store unloading
$C \leadsto c$	configuration unloading

(Value ι) (Value Fun)

$$\frac{}{\iota \leadsto \iota} \qquad \frac{S \leadsto s \quad x \notin dom(S) \quad fv(b) \subseteq dom(S) \cup \{x\} \quad locs(b) = \varnothing}{(S, \lambda(x)b) \leadsto \lambda(x)(b\{\!\{s\}\!\})}$$

(Stack []) (Stack Object)

$$\frac{}{[] \leadsto []} \qquad \frac{V \leadsto v \quad x \notin dom(S) \quad S \leadsto s}{((x \mapsto V) :: S) \leadsto ({}^{v}\!/\!{}_{x} :: s)}$$

(Object Unload) (where ℓ_i distinct)

$$\frac{S_i \leadsto s_i \quad x_i \notin dom(S_i) \quad fv(b_i) \subseteq dom(S_i) \cup \{x_i\} \quad locs(b_i) = \varnothing \quad \forall i \in 1..n}{[\ell_i = (S_i, \varsigma(x_i)b_i)\ ^{i \in 1..n}] \leadsto [\ell_i = \varsigma(x_i)(b_i\{\!\{s_i\}\!\})\ ^{i \in 1..n}]}$$

(Store Unload) (where $\Sigma = [\iota_i \mapsto O_i\ ^{i \in 1..n}]$, ι_i distinct)

$$\frac{O_i \leadsto o_i \quad \forall i \in 1..n}{\Sigma \leadsto [\iota_i \mapsto o_i\ ^{i \in 1..n}]}$$

(Config Initial) (Config Terminal)

$$\frac{S \leadsto s \quad \Sigma \leadsto \sigma \quad fv(a) \subseteq dom(S) \quad locs(a) = \varnothing}{((S, a), \Sigma) \leadsto (a\{\!\{s\}\!\}, \sigma)} \qquad \frac{V \leadsto v \quad \Sigma \leadsto \sigma}{(V, \Sigma) \leadsto (v, \sigma)}$$

We later need the following properties of the unloading relations:

Proposition 2.10

(1) *Whenever $V \leadsto v$, v is a closed value.*

(2) *Whenever $S \leadsto s$ there are distinct variables x_i and closed values v_i such that $s = [{}^{v_i}\!/\!{}_{x_i}\ ^{i \in 1..n}]$ and $dom(S) = \{x_i\ ^{i \in 1..n}\}$.*

(3) *Whenever $O \leadsto o$, object o is closed.*

(4) *Whenever $\Sigma \leadsto \sigma$, both $dom(\Sigma) = dom(\sigma)$ and $\vdash \sigma\ ok$.*

(5) *Whenever* $C \rightsquigarrow c$, $\vdash c$ *ok.*

Proof By simultaneous induction on the derivation of the unloading predicates. $\square$

The side conditions concerning free and bound variables in (Value Fun), (Stack Object), (Object Unload) and (Config Initial) are needed to ensure property (2). This property allows the substitutions that arise from closures to be manipulated easily in later proofs. All the terms manipulated by the closure-based evaluator are static terms; the side conditions concerning locations in (Value Fun), (Object Unload) and (Config Initial) ensure that only static terms arise in configurations.

We consider a store Σ to be well formed if and only if there is a store σ such that $\Sigma \rightsquigarrow \sigma$. Similarly, we consider a configuration C to be well formed if and only if there is a configuration c such that $C \rightsquigarrow c$. Notice that only static terms occur in a well formed configuration; the only occurrences of locations in a well formed configuration are in the domain of the store and in the range of any stacks occuring in the configuration.

The unloading relations are in fact functional:

Proposition 2.11 *Whenever* $\phi \rightsquigarrow \psi'$ *and* $\phi \rightsquigarrow \psi''$, *then* $\psi' = \psi''$.

Proof By induction on the derivation of $\phi \rightsquigarrow \psi'$. The only interesting cases are (Config Initial) and (Config Terminal).

(Config Initial) Here $\phi = ((S, a), \Sigma)$ and $\psi' = (a\{\!\{s'\}\!\}, \sigma')$ where $S \rightsquigarrow s'$, $\Sigma \rightsquigarrow \sigma'$, $fv(a) \subseteq dom(S)$ and $locs(a) = \varnothing$. The derivation of $\phi \rightsquigarrow \phi''$ can only have used (Config Initial) or (Config Terminal). In the former case $\psi' = \psi''$ follows easily from the induction hypothesis. The latter case can only arise when ϕ is a terminal configuration, that is, a is of the form $\lambda(x)b$. We have $\psi'' = (v'', \sigma'')$ where $\lambda(x)b \rightsquigarrow v''$ and $\Sigma \rightsquigarrow \sigma''$. The former judgment can only arise from (Value Fun). Taking alpha-conversion into account, we may assume there is a variable $x' \notin fv(b) - \{x\}$, so that $\lambda(x)b = \lambda(x')(b\{\!\{x'/x\}\!\})$ and that $\lambda(x)b \rightsquigarrow v'' = \lambda(x')(b\{\!\{x'/x\}\!\}\{\!\{s''\}\!\})$ derives by (Value Fun) from $S \rightsquigarrow s''$ given that $x' \notin dom(S)$, $fv(b\{\!\{x'/x\}\!\}) \subseteq dom(S) \cup \{x'\}$ and $locs(b\{\!\{x'/x\}\!\}) = \varnothing$. By induction hypothesis, $\sigma' = \sigma''$ and $s' = s''$. By Proposition 2.10(2), there are distinct x_i and closed values v_i such that $s' = [v_i/x_i\ {}^{i\in 1..n}]$ and $dom(S) = \{x_i\ {}^{i\in 1..n}\}$. Since $x' \notin dom(S)$, $x' \neq x_i$ for each i. Therefore we can calculate the following,

$$\begin{aligned} v'' &= \lambda(x')(b\{\!\{x'/x\}\!\}\{\!\{v_i/x_i\ {}^{i\in 1..n}\}\!\}) \\ &= \lambda(x')(b\{\!\{x'/x\}\!\})\{\!\{v_i/x_i\ {}^{i\in 1..n}\}\!\} \\ &= a\{\!\{s'\}\!\} \end{aligned}$$

which shows that $\psi'' = (v'', \sigma'') = (a\{\!\{s'\}\!\}, \sigma') = \psi'$, as required.

Case (Config Terminal) is similar. The other cases are simpler. $\square$

To prove Theorem 2.14, which asserts the consistency of the two big-step operational semantics, we need the following two lemmas.

Lemma 2.12 *If $C \leadsto c$ and $C \Downarrow C'$ there is c' such that $C' \leadsto c'$ and $c \Downarrow c'$.*

Proof By induction on the derivation of $C \Downarrow C'$. We show three typical cases.

(Closure Select) Here $C = ((S, a.\ell_j), \Sigma_0)$, $C' = (V, \Sigma_2)$ and we have

$$\begin{aligned} ((S, a), \Sigma_0) \quad &\Downarrow \quad (\iota, \Sigma_1) && (3)\\ \Sigma_1(\iota) \quad &= \quad [\ell_i = (S_i, \varsigma(x_j)b_i)^{\ i\in 1..n}] && (4)\\ (((x_j \mapsto \iota) :: S_j, b_j), \Sigma_1) \quad &\Downarrow \quad (V, \Sigma_2) && (5) \end{aligned}$$

with $j \in 1..n$ and $x_j \notin dom(S_j)$. From $C \leadsto c$ it follows there is σ_0 and s such that $\Sigma_0 \leadsto \sigma_0$, $S \leadsto s$ and $c = (a\{\!\{s\}\!\}.\ell_j, \sigma_0)$. So $((S, a), \Sigma_0) \leadsto (a\{\!\{s\}\!\}, \sigma_0)$. By the induction hypothesis and (3) there is c'_1 such that

$$(a\{\!\{s\}\!\}, \sigma_0) \Downarrow c'_1 \tag{6}$$

and $(\iota, \Sigma_1) \leadsto c'_1$. From the latter, there must be σ_1 with $\Sigma_1 \leadsto \sigma_1$ and $c'_1 = (\iota, \sigma_1)$. From (4) we know that $\iota \in dom(\Sigma_1)$; from $\Sigma_1 \leadsto \sigma_1$, it follows that $\iota \in dom(\sigma_1)$ and $\Sigma_1(\iota) \leadsto \sigma_1(\iota)$. It must be then that $\Sigma_1(\iota) \leadsto \sigma_1(\iota)$, using (Object Unload). Given (4), for each $i \in 1..n$ there is s_i such that $S_i \leadsto s_i$ and

$$\sigma_1(\iota) = [\ell_i = \varsigma(x_i)(b_i\{\!\{s_i\}\!\})^{\ i\in 1..n}] \tag{7}$$

Therefore $(((x_j \mapsto \iota) :: S_j, b_j), \Sigma_1) \leadsto (b_j\{\!\{\iota/x_j\}\!\}\{\!\{s_j\}\!\}, \sigma_1)$. Since $x_j \notin dom(S_j)$ and $S_j \leadsto s_j$, $b_j\{\!\{\iota/x_j\}\!\}\{\!\{s_j\}\!\} = b_j\{\!\{s_j\}\!\}\{\!\{\iota/x_j\}\!\}$. By the induction hypothesis and (5) there is c' such that

$$(b_i\{\!\{s_j\}\!\}\{\!\{\iota/x_j\}\!\}, \sigma_1) \Downarrow c' \tag{8}$$

and $(V, \Sigma_2) \leadsto c'$. Finally, by (Subst Select) we may derive $c \Downarrow c'$ using (6), (7) and (8).

(Closure Object) Here $C = ((S, a), \Sigma_0)$ and $C' = (\iota, \Sigma_1)$ with $a = [\ell_i = \varsigma(x_i)b_i^{\ i\in 1..n}]$, $\iota \notin dom(\Sigma_0)$, no $x_i \in dom(S)$ and

$$\Sigma_1 = (\iota \mapsto [\ell_i = (S, \varsigma(x_i)b_i)^{\ i\in 1..n}]) :: \Sigma_0.$$

So $c = (a\{\!\{s\}\!\}, \sigma_0)$ where $\Sigma_0 \rightsquigarrow \sigma_0$ and $S \rightsquigarrow s$. Therefore we can derive $c \Downarrow c'$ where $c' = (\iota, \sigma_1)$ and

$$\sigma_1 = (\iota \mapsto [\ell_i = \varsigma(x_i)(b_i\{\!\{s\}\!\})^{\ i \in 1..n}]) :: \sigma_0$$

and $\Sigma_1 \rightsquigarrow \sigma_1$.

(Closure x) Here $C = ((S, x), \Sigma)$ and $C' = (V, \Sigma)$, with $S(x) = V$. From $C \rightsquigarrow c$ it follows that $c = (v, \sigma)$ with $\Sigma \rightsquigarrow \sigma$,and $S(x) \rightsquigarrow v$. So set $c' = c$ and we have $c \Downarrow c'$ and $C' \rightsquigarrow c'$.

The other cases are similar. □

Lemma 2.13 *Suppose C is an initial configuration. Whenever $C \rightsquigarrow c$ and $c \Downarrow c'$ there is terminal C' such that $C' \rightsquigarrow c'$ and $C \Downarrow C'$.*

Proof By induction on the derivation of $c \Downarrow c'$. Either the term in C is a variable, x say, or not. If so, suppose $C = ((S, x), \Sigma)$. We must have $S \rightsquigarrow s$ and $\Sigma \rightsquigarrow \sigma$ with $x \in dom(S)$, and say $S(x) = V \rightsquigarrow v$, so that $c = (v, \sigma) = c'$. By (Closure x) we have $((S, x), \Sigma) \Downarrow (V, \Sigma)$ as required. Otherwise, the term in C is not a variable and exactly one of the (Subst $-$) rules applies. Each needs to be considered in turn; we show just one case.

(Subst Select) Here $c = (a.\ell_j, \sigma_0)$ and $c' = (v, \sigma_2)$ such that

$$\begin{array}{rcll} (a, \sigma_0) & \Downarrow & (\iota, \sigma_1) & (9) \\ \sigma_1(\iota) & = & [\ell_i = \varsigma(x_i)b_i^{\ i \in 1..n}] & (10) \\ (b_j\{\!\{\iota/x_j\}\!\}, \sigma_1) & \Downarrow & c' & (11) \end{array}$$

with $j \in 1..n$. From $C \rightsquigarrow c$ it follows that $C = ((S, a'.\ell_j), \Sigma_0)$ with $S \rightsquigarrow s$, $\Sigma_0 \rightsquigarrow \sigma_0$ and $a = a'\{\!\{s\}\!\}$. By induction hypothesis and (9), there is terminal C_1 such that

$$((S, a'), \Sigma_0) \Downarrow C_1 \tag{12}$$

and $C_1 \rightsquigarrow (\iota, \sigma_1)$. We must have $C_1 = (\iota, \Sigma_1)$ with $\Sigma_1 \rightsquigarrow \sigma_1$. By (10), $\Sigma_1(\iota) \rightsquigarrow [\ell_i = \varsigma(x_i)b_i^{\ i \in 1..n}]$ and therefore

$$\Sigma_1(\iota) = [\ell_i = (S_i, \varsigma(x_i)b'_i)^{\ i \in 1..n}] \tag{13}$$

with $S_j \rightsquigarrow s_j$, $b_j = b'_j\{\!\{s_j\}\!\}$ and $x_j \notin dom(S_j)$. Now since we may derive $(((x_j \mapsto \iota) :: S_j, b'_j), \Sigma_1) \rightsquigarrow (b'_j\{\!\{\iota/x_j\}\!\}, \sigma_1)$, the induction hypothesis and (11) imply there is C' with

$$(((x_j \mapsto \iota) :: S_j, b'_j), \Sigma_1) \Downarrow C' \tag{14}$$

and $C' \rightsquigarrow c'$. Combining (12), (13) and (14) using (Closure Select) we obtain $C \Downarrow C'$ as required.

The other cases are similar. □

Theorem 2.14 *Suppose C and C' are initial and terminal configurations respectively, and that $C \rightsquigarrow c$ and $C' \rightsquigarrow c'$. Then $C \Downarrow C'$ if and only if $c \Downarrow c'$.*

Proof Suppose $C \Downarrow C'$. By Lemma 2.12 there is c'' with $C' \rightsquigarrow c''$ and $c' \Downarrow c''$. By Proposition 2.11, $c' = c''$. On the other hand, suppose $c \Downarrow c'$. By Lemma 2.13, there is a terminal configuration C'' such that $C'' \rightsquigarrow c'$ and $C \Downarrow C''$. By Proposition 2.11, $C' = C''$. □

2.5 Discussion and Related Work

A big-step closure-based semantics, as in Section 2.4 or, say, the definition of Standard ML, is attractive as a language definition because it directly yields an efficient algorithm for interpreting the calculus. For instance, Cardelli (1995) implements Obliq in this way. On the other hand, substitution-based semantics are simpler to work with when reasoning about program equivalence; we apply the substitution-based semantics of Sections 2.2 and 2.3 in Sections 4 and 5 respectively. In fact, either substitution-based semantics would do alone; we include both for the sake of completeness.

We do not present a small-step closure-based semantics for the imperative object calculus; this would amount to an SECD machine (Landin 1964) for the calculus. The next section, however, contains a small-step closure-based semantics for an object-oriented abstract machine to which we compile the object calculus.

The technique used to prove Theorem 2.7, the consistency of the two substitution-based semantics is well-known. An analogous result is proved by Plotkin (1975), who also proves the consistency with the SECD machine of what amounts to a big-step substitution-based operational semantics. On the other hand, the proof technique of Theorem 2.14, the consistency of the substitution-based and closure-based big-step semantics, appears to be new, though the idea of unloading a closure to a term goes back to Plotkin (1975). Felleisen (1995) proves correspondences between a variety of small-step semantics, both substitution-based and closure-based, including a result analogous to Theorem 2.7.

3 Compilation to an Abstract Machine

In this section we present an abstract machine, based on the ZAM (Leroy 1990), for the extended calculus of imperative objects, a compiler sending the object calculus to the instruction set of the abstract machine and a proof of correctness. The proof depends on an unloading procedure which converts configurations of the abstract machine back into configurations of the object calculus from Section 2. The unloading procedure depends on a modified abstract machine whose argument stack and environment contain object calculus terms as well as locations.

3.1 The Abstract Machine

The machine defined here is based on Leroy's ZAM. The ZAM mechanism for function calls is used, so that to call a function a mark is pushed onto the stack, the arguments are evaluated and pushed onto the stack and the code for the function body is called. The body of the function can read in its (curried) arguments off the stack, and discovers when it has consumed all its arguments when it finds the mark. When the function returns (on executing a `return` instruction) if there are more arguments to consume, the result of the function (which must itself be a function if execution is to proceed) is called, and will consume the extra arguments that are available.

The instruction set of our abstract machine consists of the following *operations*.

op ::=	operation
`access` i	variable access
`object`$[(\ell_i, ops_i)\ ^{i \in 1..n}]$	object construction
`select` ℓ	method invocation
`update`(ℓ, ops)	method update
`let` ops	let
`cur` ops	build function closure
`apply` ops	apply function
`grab`	get curried argument
`pushmark`	push mark onto stack
`return`	return from function
ops ::= $[] \mid op :: ops$	

We use the notation $\texttt{grab}^n$ for the list $[\texttt{grab}, \texttt{grab}, \ldots, \texttt{grab}]$ consisting of n `grab` instructions. We represent compilation of a term a to an operation list ops by the judgment $xs \vdash a \Rightarrow ops$, defined by the following rules. The variable list xs includes all the free variables of a; it is needed to compute the de Bruijn index of each variable occurring in a.

(Trans Var) $[x_i\ ^{i \in 1..n}] \vdash x_j \Rightarrow [\texttt{access}\ j]$ if $j \in 1..n$.

(Trans Object) $xs \vdash [\ell_i = \varsigma(y_i)a_i \ ^{i \in 1..n}] \Rightarrow [\texttt{object}[(\ell_i, ops_i)\ ^{i \in 1..n}]]$
if $y_i :: xs \vdash a_i \Rightarrow ops_i$ and $y_i \notin xs$ for all $i \in 1..n$.

(Trans Select) $xs \vdash a.\ell \Rightarrow ops@[\texttt{select}\ \ell]$ if $xs \vdash a \Rightarrow ops$.

(Trans Update) $xs \vdash (a.\ell \Leftarrow \varsigma(x)a') \Rightarrow ops@[\texttt{update}(\ell, ops')]$
if $xs \vdash a \Rightarrow ops$ and $x :: xs \vdash a' \Rightarrow ops'$ and $x \notin xs$.

(Trans Clone) $xs \vdash clone(a) \Rightarrow ops@[\texttt{clone}]$ if $xs \vdash a \Rightarrow ops$.

(Trans Let) $xs \vdash let\ x = a\ in\ a' \Rightarrow ops@[\texttt{let}(ops')]$
if $xs \vdash a \Rightarrow ops$ and $x :: xs \vdash a' \Rightarrow ops'$ and $x \notin xs$.

(Trans Apply) $xs \vdash (a_1 a_2 \ldots a_n) \Rightarrow \texttt{pushmark} :: ops_n @ ops_{n-1} @ \ldots @ ops_1 @[\texttt{apply}]$
if $xs \vdash a_i \Rightarrow ops_i$ for all $i \in 1..n$ and a_1 is not a function application.

(Trans Function) $xs \vdash \lambda(x_1 x_2 \ldots x_{n+1})a \Rightarrow [\texttt{cur}(\texttt{grab}^n @ ops @ [\texttt{return}])]$
if $x_i \notin xs$ for all $i \in 1..n+1$, all the x_i are distinct, a is not a λ abstraction and $[x_i\ ^{i \in 1..n+1}]@xs \vdash a \Rightarrow ops$.

An abstract machine *configuration*, C or D, is a pair (P, Σ), where P is a state and Σ is a store, given as follows:

$P, Q ::= (ops, E, AS, RS)$	machine state
$U, V ::= \iota \mid fun(ops, E)$	value
$U^\diamondsuit, V^\diamondsuit ::= U \mid \diamondsuit$	value or mark
$E ::= [U_i\ ^{i \in 1..n}]$	environment
$AS ::= [U_i^{\diamondsuit}\ ^{i \in 1..n}]$	argument stack
$RS ::= [F_i\ ^{i \in 1..n}]$	return stack
$F ::= (ops, E)$	closure
$O ::= [(\ell_i, F_i)\ ^{i \in 1..n}]$	stored object (ℓ_i distinct)
$\Sigma ::= [\iota_i \mapsto O_i\ ^{i \in 1..n}]$	store (ι_i distinct)

In a configuration $((ops, E, AS, RS), \Sigma)$, ops is the current program. Environment E contains variable bindings. Argument stack AS contains results of evaluating terms and control flow information in the form of marks, $\diamondsuit$. Return stack RS holds return addresses during function calls and method invocations. Store Σ associates locations with objects.

Two transition relations, given next, represent execution of the abstract machine. A β*-transition*, $P \xrightarrow{\beta} Q$, corresponds directly to a reduction in the object calculus. A τ*-transition*, $P \xrightarrow{\tau} Q$, is an internal step of the abstract machine, for example a method return or a variable lookup. Lemma 3.12 relates reductions of the object calculus and transitions of the abstract machine.

(τ Return) $(([], E, AS, (ops, E') :: RS), \Sigma) \xrightarrow{\tau} ((ops, E', AS, RS), \Sigma)$.

(τ Function Return) $(([\texttt{return}], E, U :: \Diamond :: AS, (ops, E') :: RS), \Sigma) \xrightarrow{\tau} ((ops, E', U :: AS, RS), \Sigma)$.

(β Function Return) $(([\texttt{return}], E, \mathit{fun}(ops, E') :: U :: AS, RS), \Sigma) \xrightarrow{\beta} ((ops, U :: E', AS, RS), \Sigma)$.

(τ Grab) $((\texttt{grab} :: ops, E, \Diamond :: AS, (ops', E') :: RS), \Sigma) \xrightarrow{\tau} ((ops', E, \mathit{fun}(ops, E) :: AS, RS), \Sigma)$.

(β Grab) $((\texttt{grab} :: ops, E, U :: AS, RS), \Sigma) \xrightarrow{\beta} ((ops, U :: E, AS, RS), \Sigma)$.

(τ Access) $((\texttt{access}\ j :: ops, E, AS, RS), \Sigma) \xrightarrow{\tau} ((ops, E, U_j :: AS, RS), \Sigma)$ if $E = [U_i\ {}^{i \in 1..n}]$ and $j \in 1..n$.

(τ Pushmark)
$((\texttt{pushmark} :: ops, E, AS, RS), \Sigma) \xrightarrow{\tau} ((ops, E, \Diamond :: AS, RS), \Sigma)$.

(τ Cur) $((\texttt{cur}\ ops :: ops', E, AS, RS), \Sigma) \xrightarrow{\tau} ((ops', E, \mathit{fun}(ops, E) :: AS, RS), \Sigma)$.

(β Clone) $((\texttt{clone} :: ops, E, \iota :: AS, RS), \Sigma) \xrightarrow{\beta} ((ops, E, \iota' :: AS, RS), \Sigma')$ if $\Sigma(\iota) = O$ and $\Sigma' = (\iota' \mapsto O) :: \Sigma$ and $\iota' \notin dom(\Sigma)$.

(β Object) $((\texttt{object}[(\ell_i, ops_i)\ {}^{i \in 1..n}] :: ops, E, AS, RS), \Sigma) \xrightarrow{\beta} ((ops, E, \iota :: AS, RS), (\iota \mapsto [(\ell_i(ops_i, E))\ {}^{i \in 1..n}]) :: \Sigma)$ if $\iota \notin dom(\Sigma)$.

(β Select) $((\texttt{select}\ \ell_j :: ops, E, \iota :: AS, RS), \Sigma) \xrightarrow{\beta} ((ops_j, \iota :: E_j, AS, (ops, E) :: RS), \Sigma)$
if $\Sigma(\iota) = [(\ell_i, (ops_i, E_i))\ {}^{i \in 1..n}]$ and $j \in 1..n$.

(β Update)
$((\texttt{update}(\ell, ops') :: ops, E, \iota :: AS, RS), \Sigma) \xrightarrow{\beta} ((ops, E, \iota :: AS, RS), \Sigma')$
if $\Sigma(\iota) = O@[(\ell, F)]@O'$ and $\Sigma' = \Sigma + (\iota \mapsto O@[(\ell, (ops', E))]@O')$.

(β Let) $((\texttt{let}\ ops' :: ops, E, U :: AS, RS), \Sigma) \xrightarrow{\beta} ((ops', U :: E, AS, (ops, E) :: RS), \Sigma)$.

(β Apply) $((\texttt{apply} :: ops, E, \mathit{fun}(ops', E') :: U :: AS, RS), \Sigma) \xrightarrow{\beta} ((ops', U :: E', AS, (ops, E) :: RS), \Sigma)$.

Let $C \xrightarrow{\beta\tau} D$ if $C \xrightarrow{\beta} D$ or $C \xrightarrow{\tau} D$.

3.2 An Example

We take as an example the term $(\lambda(x)x.\ell)[\ell = \varsigma(s)\lambda(y)y][]$ which compiles to

$$ops = [\texttt{pushmark}, \texttt{object}[], \texttt{object}[(\ell, \texttt{cur}(ops_1))], \texttt{cur}(ops_2), \texttt{apply}]$$

where

$$\begin{aligned} ops_1 &= [\texttt{access}\ 1, \texttt{return}] \\ ops_2 &= [\texttt{access}\ 1, \texttt{select}\ \ell, \texttt{return}] \end{aligned}$$

Let $\Sigma_1 = [\iota_1 \mapsto []]$, $\Sigma_2 = [\iota_1 \mapsto [], \iota_2 \mapsto [(\ell, (\texttt{cur}(ops_1), []))]]$ and $RS = [([\texttt{return}], [\iota_2]), ([], [])]$. The program ops_1 results in the following computation:

$$\begin{array}{l} ((ops, [], [], []), []) \\ \quad \xrightarrow{\tau} \ (([\texttt{object}[], \texttt{object}[(\ell, \texttt{cur}(ops_1))], \texttt{cur}(ops_2), \texttt{apply}], [], [\diamondsuit], []), []) \\ \quad \xrightarrow{\beta} \ (([\texttt{object}[(\ell, \texttt{cur}(ops_1))], \texttt{cur}(ops_2), \texttt{apply}], [], [\iota_1, \diamondsuit], []), \Sigma_1) \\ \quad \xrightarrow{\beta} \ (([\texttt{cur}(ops_2), \texttt{apply}], [], [\iota_2, \iota_1, \diamondsuit], []), \Sigma_2) \\ \quad \xrightarrow{\tau} \ (([\texttt{apply}], [], [\mathit{fun}(ops_2, []), \iota_2, \iota_1, \diamondsuit], []), \Sigma_2) \\ \quad \xrightarrow{\beta} \ ((ops_2, [\iota_2], [\iota_1, \diamondsuit], [([], [])]), \Sigma_2) \\ \quad \xrightarrow{\tau} \ (([\texttt{select}\ \ell, \texttt{return}], [\iota_2], [\iota_2, \iota_1, \diamondsuit], [([], [])]), \Sigma_2) \\ \quad \xrightarrow{\beta} \ (([\texttt{cur}(ops_1)], [\iota_2], [\iota_1, \diamondsuit], RS), \Sigma_2) \\ \quad \xrightarrow{\tau} \ (([], [\iota_2], [\mathit{fun}(ops_1, [\iota_2]), \iota_1, \diamondsuit], RS), \Sigma_2) \\ \quad \xrightarrow{\tau} \ (([\texttt{return}], [\iota_2], [\mathit{fun}(ops_1, [\iota_2]), \iota_1, \diamondsuit], [([], [])]), \Sigma_2) \\ \quad \xrightarrow{\beta} \ ((ops_1, [\iota_1, \iota_2], [\diamondsuit], [([], [])]), \Sigma_2) \\ \quad \xrightarrow{\tau} \ (([\texttt{return}], [\iota_1, \iota_2], [\iota_1, \diamondsuit], [([], [])], \Sigma_2) \\ \quad \xrightarrow{\tau} \ (([], [], [\iota_1], []), \Sigma_2) \end{array}$$

$(([], [], [\iota_1], []), \Sigma_2)$ is the terminal configuration of the abstract machine. When the abstract machine terminates, the answer to the computation can be found as the single item on the argument stack. In this case, the argument stack contains ι_1, the location of $[]$ in the store Σ_2.

3.3 The Unloading Machine

To prove the abstract machine and compiler correct, we need to convert back from a machine state to an object calculus term. To do so, we load the state into a modified abstract machine, the *unloading machine*, and when this unloading machine terminates, its argument stack contains a single term corresponding to the original machine state.

The unloading machine is like the abstract machine, except that instead of executing each instruction, it reconstructs the corresponding source term. Since no store lookups or updates are performed, the unloading machine does not act on a store. An unloading machine state is like an abstract machine state, except that values are generalised to arbitrary terms. Let an *unloading machine state*, p or q, be a quadruple (ops, e, as, RS) where e and as are defined as follows:

$e ::= [a_i \,^{i \in 1..n}]$	unloading environment
$a^\diamondsuit, b^\diamondsuit ::= a \mid \diamondsuit$	term or mark
$as ::= [a_i^\diamondsuit \,^{i \in 1..n}]$	unloading stack

Next we make a simultaneous inductive definition of a *u-transition* relation $p \xrightarrow{u} p'$, an unloading relation, $(ops, e) \rightsquigarrow (x)b$, that unloads a method closure to a method, $fun(ops, e) \rightsquigarrow \lambda(x)b$ that unloads a closure to a λ-abstraction and $[U_i^\diamondsuit \,^{i \in 1..n}] \rightsquigarrow [a_i^\diamondsuit \,^{i \in 1..n}]$ that unloads a list.

(u **Access**) $(\texttt{access}\ j :: ops', e, as, RS) \xrightarrow{u} (ops', e, a_j :: as, RS)$
if $j \in 1..n$ and $e = [a_i \,^{i \in 1..n}]$.

(u **Object**) $(\texttt{object}[(\ell_i, ops_i) \,^{i \in 1..n}] :: ops', e, as, RS) \xrightarrow{u}$
$(ops', e, [\ell_i = \varsigma(x_i)b_i{}^{i \in 1..n}] :: as, RS)$ if $(ops_i, e) \rightsquigarrow (x_i)b_i$ for each $i \in 1..n$.

(u **Clone**) $(\texttt{clone} :: ops', e, a :: as, RS) \xrightarrow{u} (ops', e, (clone(a)) :: as, RS)$.

(u **Select**) $(\texttt{select}\ \ell :: ops', e, a :: as, RS) \xrightarrow{u} (ops', e, (a.\ell) :: as, RS)$.

(u **Update**) $(\texttt{update}(\ell, ops) :: ops', e, a :: as, RS) \xrightarrow{u}$
$(ops', e, (a.\ell \Leftarrow \varsigma(x)b) :: as, RS)$ if $(ops, e) \rightsquigarrow (x)b$.

(u **Let**) $(\texttt{let}(ops') :: ops'', e, a :: as, RS) \xrightarrow{u} (ops'', e, (let\ x = a\ in\ b) :: as, RS)$
if $(ops', e) \rightsquigarrow (x)b$.

(u **Return**) $([], e, as, (ops, E) :: RS) \xrightarrow{u} (ops, e', as, RS)$
if $E \rightsquigarrow e'$.

(u Cur) $(\texttt{cur}\ ops :: ops', e, as, RS) \xrightarrow{u} (ops', e, (\lambda(x)a) :: as, RS)$ if $(ops, x :: e, [\Diamond], []) \rightsquigarrow a$ and $x \notin fv(e)$.

(u Function Return) $([\texttt{return}], e, [a_i\ ^{i\in 1..n}]@[\Diamond]@as, (ops, E') :: RS) \xrightarrow{u} (ops, e', (a_1 a_2 \ldots a_n) :: as, RS)$ if $E' \rightsquigarrow e'$.

(u Grab) $(\texttt{grab} :: ops, e, as, RS) \xrightarrow{u} ([], e, (\lambda(x)a) :: as, RS)$ if $fun(ops, e) \rightsquigarrow \lambda(x)a$.

(u Apply) $(\texttt{apply} :: ops, e, [a_i\ ^{i\in 1..n}]@[\Diamond]@as, RS) \xrightarrow{u} (ops, e, (a_1 a_2 \ldots a_n) :: as, RS)$.

(Unload Abstraction) $(ops, e) \rightsquigarrow (x)b$ if $x \notin fv(e)$ and $(ops, x :: e, [], []) \xrightarrow{u}{}^{*} ([], e', [b], [])$.

(Unload Closure) $fun(ops, e) \rightsquigarrow \lambda(x)b$ if $x \notin fv(e)$ and $(ops, x :: e, [\Diamond], []) \xrightarrow{u}{}^{*} ([], e', [b], [])$.

(Unload List Empty) $[] \rightsquigarrow []$.

(Unload List Loc) $\iota :: [U_i^{\Diamond}\ ^{i\in 1..n}] \rightsquigarrow \iota :: [a_i^{\Diamond}\ ^{i\in 1..n}]$ if $[U_i^{\Diamond}\ ^{i\in 1..n}] \rightsquigarrow [a_i^{\Diamond}\ ^{i\in 1..n}]$.

(Unload List Closure) $fun(ops, E) :: [U_i^{\Diamond}\ ^{i\in 1..n}] \rightsquigarrow (\lambda(x)a) :: [a_i^{\Diamond}\ ^{i\in 1..n}]$ if $[U_i^{\Diamond}\ ^{i\in 1..n}] \rightsquigarrow [a_i^{\Diamond}\ ^{i\in 1..n}]$ and $fun(ops, E) \rightsquigarrow \lambda(x)a$.

(Unload List Mark) $\Diamond :: [U_i^{\Diamond}\ ^{i\in 1..n}] \rightsquigarrow \Diamond :: [a_i^{\Diamond}\ ^{i\in 1..n}]$ if $[U_i^{\Diamond}\ ^{i\in 1..n}] \rightsquigarrow [a_i^{\Diamond}\ ^{i\in 1..n}]$.

We complete the machine with the following unloading relations: $O \rightsquigarrow o$ (on objects), $\Sigma \rightsquigarrow \sigma$ (on stores) and $C \rightsquigarrow c$ (on configurations).

(Unload Object) $[(\ell_i, (ops_i, E_i))\ ^{i\in 1..n}] \rightsquigarrow [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$ if $(ops_i, E_i) \rightsquigarrow (x_i)b_i$ for all $i \in 1..n$.

(Unload Store) $[\iota_i \mapsto O_i\ ^{i\in 1..n}] \rightsquigarrow [\iota_i \mapsto o_i\ ^{i\in 1..n}]$ if $O_i \rightsquigarrow o_i$ for all $i \in 1..n$.

(Unload Config) $((ops, E, AC, RS), \Sigma) \rightsquigarrow (a, \sigma)$ if $\Sigma \rightsquigarrow \sigma$ and $(ops, E, AC, RS) \xrightarrow{u}{}^{*} ([], e', [a], [])$.

Let $p \rightsquigarrow a$ if and only if there is e such that $p \xrightarrow{u}{}^{*} ([], e, [a], [])$. We say $P \downarrow p$ if $P = (ops, E, AS, RS)$, $p = (ops, e, as, RS)$, $E \rightsquigarrow e$ and $AS \rightsquigarrow as$. Therefore $(P, \Sigma) \rightsquigarrow (a, \sigma)$ if and only if $P \downarrow p$, $p \rightsquigarrow a$ and $\Sigma \rightsquigarrow \sigma$. Define the *shape* of (ops, e, as', RS) to be the quadruple $(ops, |e|, |as|, RS)$, and write shape p for the shape of p. We say two stacks $[a_i^{\diamondsuit}\ {}^{i\in 1..n}]$ and $[b_i^{\diamondsuit}\ {}^{i\in 1..m}]$ are *mark-equivalent* if and only if $n = m$ and $a_j^{\diamondsuit} = \diamondsuit$ if and only if $b_j^{\diamondsuit} = \diamondsuit$. We say p and q are *shape-mark-equivalent* if shape p = shape q and the argument stack of p is mark-equivalent to that of q.

3.4 Correctness of the Abstract Machine

We start with a lemma which shows that the unloading machine is independent of the terms in its environment and on its stack.

Lemma 3.1 *If $p \xrightarrow{u} p'$ and p is shape-mark-equivalent to q then there is a q' with $q \xrightarrow{u} q'$ and p' is shape-mark-equivalent to q'.*

Proof This is proved by induction on the derivation of $p \xrightarrow{u} p'$. For example, if $p = (\texttt{let}\ ops' :: ops'', e, a :: as, RS)$ then $p' = (ops'', e, [let\ x = a\ in\ b] :: as, RS)$ where $(ops'', e) \rightsquigarrow (x)b$. Let $q = (\texttt{let}\, ops' {::} ops'', e', a' {::} as', RS)$ where $|e'| = |e|$, $|as| = |as'|$ and as is mark-equivalent to as'. $(ops'', e) \rightsquigarrow (x)b$ means that there are p_j for $j \in 1..n$ with $p_1 = (ops'', x :: e, [], [])$, $p_n = ([], e'', [b], [])$ and for all $j \in 1..n-1$ $p_j \xrightarrow{u} p_{j+1}$. Then we can apply the inductive hypothesis to get q_i for $j \in 1..n$ with $q_1 = (ops'', x :: e', [], [])$, $q_n = ([], e''', [b'], [])$ and for all $j \in 1..n-1$ $q_j \xrightarrow{u} q_{j+1}$. Hence $(ops'', e') \rightsquigarrow (x)b'$. By ($u$ Let) $q \xrightarrow{u} q' = (ops'', e', [let\ x = a'\ in\ b'] :: as', RS)$ and p' is shape-mark-equivalent to q'. □

A corollary of Lemma 3.1 is the following:

Lemma 3.2 *If $p \rightsquigarrow a$ then for all q with p and q shape-mark-equivalent, there is an a' with $q \rightsquigarrow a'$.*

Proof $p \rightsquigarrow a$ means that there is a sequence of reductions

$$p \xrightarrow{u} p_1 \xrightarrow{u} \cdots \xrightarrow{u} p_n$$

where p_n is of the form $([], e, [a], [])$. Applying Lemma 3.1 inductively to this chain gives a new chain of reductions

$$q \xrightarrow{u} q_1 \xrightarrow{u} \cdots \xrightarrow{u} q_n$$

where q_n is of the form $([], e', [a'], [])$, and hence $q \rightsquigarrow a'$. □

The following lemma describes some of the behaviour of the stacks of the unloading machine.

Lemma 3.3

(1) *If* $(ops, e, as, []) \xrightarrow{u} (ops', e', as', [])$ *then for all* as'' *and for all* RS, $(ops, e, as@as'', RS) \xrightarrow{u} (ops', e', as'@as'', RS)$.

(2) *For all ops,* $([op], e, as, []) \xrightarrow{u} ([], e', as', [])$ *if and only if we have the transition* $(op :: ops, e, as, []) \xrightarrow{u} (ops, e', as', [])$.

(3) *If we have the chains of transitions* $(ops, e, as, []) \xrightarrow{u}{}^{*} ([], e_1, as_1, [])$ *and* $(ops_1, e_1, as_1, []) \xrightarrow{u}{}^{*} ([], e_2, as_2, [])$ *then we can compose them, getting* $(ops@ops_1, e, as, []) \xrightarrow{u}{}^{*} ([], e_2, as_2, [])$.

Proof Inspecting the rules for u transitions gives (1) and (2). To prove (3), we note that no u-transition increases RS, and induct on the length of ops, applying (2). $\square$

We aim to show that unloading is an inverse to compilation. We prove a more general fact first.

Lemma 3.4 *If* $x_i{}^{i\in 1..n} \vdash a \Rightarrow ops$ *then for all* $b_i{}^{i\in 1..n}$, $(ops, [b_i{}^{i\in 1..n}], [], []) \rightsquigarrow a\{\!\{b_i/x_i{}^{i\in 1..n}\}\!\}$.

Proof We prove this by induction on the derivation of $x_i{}^{i\in 1..n} \vdash a \Rightarrow ops$, considering each of the Trans rules individually.

(Trans Var) Here $a = x_j$, where $j \in 1..n$. Then $x_i{}^{i\in 1..n} \vdash a \Rightarrow [\texttt{access}\ j]$ and $([\texttt{access}\ j], [b_i{}^{i\in 1..n}], [], []) \xrightarrow{u} ([], [b_i{}^{i\in 1..n}], [b_j], [])$

(Trans Select) Here $a = a'.\ell$. We have an ops' with $x_i{}^{i\in 1..n} \vdash a' \Rightarrow ops'$. Then $x_i{}^{i\in 1..n} \vdash a \Rightarrow ops'@[\texttt{select}\ \ell]$. By rule induction we have that $(ops', [b_i{}^{i\in 1..n}], [], []) \xrightarrow{u}{}^{*} ([], [b_i{}^{i\in 1..n}], [a''], [])$, where $a'' = a'\{\!\{b_i/x_i{}^{i\in 1..n}\}\!\}$. By Lemma 3.3(3) we know $(ops'@[\texttt{select}\ \ell], [b_i{}^{i\in 1..n}], [], []) \xrightarrow{u}{}^{*} p = ([\texttt{select}\ \ell], [b_i{}^{i\in 1..n}], [a''], [])$, and $p \xrightarrow{u} ([], [b_i{}^{i\in 1..n}], [a''.\ell], [])$. This suffices, since $a''.\ell = (a'\{\!\{b_i/x_i{}^{i\in 1..n}\}\!\}).\ell = (a'.\ell)\{\!\{b_i/x_i{}^{i\in 1..n}\}\!\}$.

(Trans Let) Here $a = (let\ x = a_1\ in\ a_2)$. We have ops_1, ops_2 with $x_i{}^{i\in 1..n} \vdash a_1 \Rightarrow ops_1$ and $x :: [x_i{}^{i\in 1..n}] \vdash a_2 \Rightarrow ops_2$ (where $x \notin x_i{}^{i\in 1..n}$). Then $x_i{}^{i\in 1..n} \vdash a \Rightarrow ops_1@[\texttt{let}(ops_2)]$. Rule induction applied to $[x_i{}^{i\in 1..n}] \vdash a_1 \Rightarrow ops_1$ gives $(ops_1, [b_i{}^{i\in 1..n}], [], []) \xrightarrow{u}{}^{*} ([], [b_i{}^{i\in 1..n}], [a_1'], [])$ where $a_1' = a_1\{\!\{b_i/x_i{}^{i\in 1..n}\}\!\}$. Rule induction applied to $x :: [x_i{}^{i\in 1..n}] \vdash a_2 \Rightarrow ops_2$ gives $(ops_2, x :: [b_i{}^{i\in 1..n}], [], []) \xrightarrow{u}{}^{*} ([], x :: [b_i{}^{i\in 1..n}], [a_2'], [])$ where $a_2' = a_2\{\!\{x/x\}\!\}\{\!\{b_i/x_i{}^{i\in 1..n}\}\!\}$. By (Unload Abstraction) $(ops_2, [b_i{}^{i\in 1..n}]) \rightsquigarrow$

$(x)a_2'$. Applying Lemma 3.3(3), $(ops_1@[\mathtt{let}(ops_2)], [b_i\ ^{i \in 1..n}], [], []) \xrightarrow{u}{}^{*}$ $([\mathtt{let}(ops_2)], [b_i\ ^{i \in 1..n}], [a_1'], []) \xrightarrow{u} ([], [b_i\ ^{i \in 1..n}], [let\ x = a_1'\ in\ a_2'], [])$ by (u Let). This is sufficient, since $(let\ x = a_1'\ in\ a_2') = a\{\!\{b_i/x_i\ ^{i \in 1..n}\}\!\}$.

(Trans Clone) Here $a = clone(a')$. This follows in the same way as the (Trans Select) case.

(Trans Update) Here $a = (a_1.\ell \Leftarrow \varsigma(x)a_2)$. Derived from $x_i\ ^{i \in 1..n} \vdash a_1 \Rightarrow ops_1$ and $x :: [x_i\ ^{i \in 1..n}] \vdash a_2 \Rightarrow ops_2$, where $x \notin xs$, we have $x_i\ ^{i \in 1..n} \vdash a \Rightarrow ops_1@[\mathtt{update}(\ell, ops_2)]$. Via reasoning similar to the case of (Trans Let), we calculate: $(ops_1@[\mathtt{update}(\ell, ops_2)], [b_i\ ^{i \in 1..n}], [], []) \xrightarrow{u}{}^{*}$ $([], [b_i\ ^{i \in 1..n}], [(a_1'\ell \Leftarrow \varsigma(x)a_2')], [])$ where $a_1' = a_1\{\!\{b_i/x_i\ ^{i \in 1..n}\}\!\}$ and $a_2' = a_2\{\!\{x :: [b_i\ ^{i \in 1..n}]/x :: [x_i\ ^{i \in 1..n}]\}\!\}$. This is sufficient, since $a\{\!\{b_i/x_i\ ^{i \in 1..n}\}\!\} = (a_1'.\ell \Leftarrow \varsigma(x)a_2')$.

(Trans Object) Here $a = [(\ell_i, \varsigma(y_i)a_i)\ ^{i \in 1..n}]$. If $y_i :: [x_i\ ^{i \in 1..n}] \vdash a_i \Rightarrow ops_i$ then $x_i\ ^{i \in 1..n} \vdash a \Rightarrow [\mathtt{object}[(\ell_i, ops_i)\ ^{i \in 1..n}]]$. By rule induction we have that for all i, $(ops_i, y_i :: [b_j\ ^{j \in 1..n}]) \rightsquigarrow (y_i)a_i'$ where $a_i' = a_i\{\!\{b_j/x_j\ ^{j \in 1..n}\}\!\}$ and hence that $([\mathtt{object}[(\ell_i, ops_i)^{i \in 1..n}]], [b_i^{i \in 1..n}], [], []) \xrightarrow{u} ([], [b_i^{i \in 1..n}], [\ell_i = \varsigma(y_i)a_i'], [])$ as required.

(Trans Function) Here $a = \lambda(y_1 \ldots y_{m+1})b$ where b is not a function, $y_i \notin [x_i\ ^{i \in 1..n}]$ for each $i \in 1..m+1$, $[y_i\ ^{i \in 1..m+1}]\ @\ xs \vdash b \Rightarrow ops$ and $[x_i\ ^{i \in 1..n}] \vdash a \Rightarrow [\mathtt{cur}(\mathtt{grab}^m\ @\ ops\ @\ [\mathtt{return}])]$. Let $bs = [b_i\ ^{i \in 1..n}]$, $xs = [x_i\ ^{i \in 1..n}]$ and $e_k = [y_{m+2-i}\ ^{i \in 1..k}]@bs$ for each $k \in 1..m+1$. By induction, $(ops, e_{m+1}, [], []) \rightsquigarrow b'$ where $b' = b\{\!\{bs/xs\}\!\}$ so $(\mathtt{grab} :: ops, e_m, [], []) \xrightarrow{u} ([], e_m, [\lambda(y_{m+1})b', []])$ and (after iterating m times) $(\mathtt{grab}^m\ @\ ops, e_{m+1}, [], []) \xrightarrow{u} ([], e_{m+1}, [\lambda(y_2 \ldots y_{m+1})b']$. We now infer $(\mathtt{grab}^m\ @\ ops\ @\ [\mathtt{return}], y_1 :: bs, [\diamondsuit], []) \xrightarrow{u}{}^{*} ([], y_1 :: bs, [a_1']), [])$ (where $a_1' = \lambda(y_2 y_3 \ldots y_{m+1})b'$]) from Lemma 3.3(3) and Lemma 3.3(1) and hence by (u Cur), $([\mathtt{cur}(\mathtt{grab}^m\ @\ ops\ @\ [\mathtt{return}])], bs, [], []) \rightsquigarrow \lambda(y_1 y_2 \ldots y_{m+1})b' = a\{\!\{bs/xs\}\!\}$.

(Trans Apply) Here $a = (a_1 a_2 \ldots a_m)$, and $[x_i\ ^{i \in 1..n}] \vdash a \Rightarrow \mathtt{pushmark} :: ops_m@ops_{m-1}@\ldots@ops_1@[\mathtt{apply}]$ where for each $j \in 1..m$ $[x_i\ ^{i \in 1..n}] \vdash a_j \Rightarrow ops_j$. The induction hypothesis says that for each $j \in 1..m$ we have $(ops_j, [b_i\ ^{i \in 1..n}], [], []) \rightsquigarrow a_j\{\!\{b_i/x_i\ ^{i \in 1..n}\}\!\}$. Lemmas 3.3(1) and 3.3(3) give that $(\mathtt{pushmark} :: ops_m@\ldots@ops_1@[\mathtt{apply}], [b_i^{i \in 1..n}], [], []) \xrightarrow{u}{}^{*} p = ([\mathtt{apply}], [b_i\ ^{i \in 1..n}], [a_1', \ldots, a_m', \diamondsuit], [])$ where $a_j' = a_j\{\!\{b_i/x_i\ ^{i \in 1..n}\}\!\}$. $p \xrightarrow{u}$ $([], [b_i\ ^{i \in 1..n}], [a'], [])$ where $a' = a\{\!\{b_i/x_i\ ^{i \in 1..n}\}\!\}$ as required. $\square$

As a corollary we have that unloading is an inverse to compilation.

Proposition 3.5 *Whenever* $[] \vdash a \Rightarrow ops$ *then* $((ops, [], [], []), []) \rightsquigarrow (a, [])$.

The unloading machine preserves substitutions.

Lemma 3.6 *If* $p \xrightarrow{u} q$ *then* $p\{\!\{a/x\}\!\} \xrightarrow{u} q\{\!\{a/x\}\!\}$.

Proof By inspecting the u-transition rules. For example, if $p = (\mathtt{access}\ j :: ops, [a_i{}^{i \in 1..n}], as, RS)$ and $p \xrightarrow{u} q = (ops, [a_i{}^{i \in 1..n}], a_j :: as, RS)$, then $p\{\!\{a/x\}\!\} = (\mathtt{access}\ j :: ops, [a_i\{\!\{a/x\}\!\}\ {}^{i \in 1..n}], as\{\!\{a/x\}\!\}, RS)$ and by (u Access), $p\{\!\{a/x\}\!\} \xrightarrow{u} q\{\!\{a/x\}\!\} = (ops, [a_i\{\!\{a/x\}\!\}\ {}^{i \in 1..n}], (a_j\{\!\{a/x\}\!\}) :: as\{\!\{a/x\}\!\}, RS)$. □

Lemma 3.7

(1) *If* $p \xrightarrow{u} q$ *then* $fv(q) \subseteq fv(p)$.

(2) *If* $(ops, e) \rightsquigarrow (x)b$ *then* $fv(b) - \{x\} \subseteq fv(e)$.

(3) *If* $fun(ops, e) \rightsquigarrow \lambda(x)b$ *then* $fv(b) - \{x\} \subseteq fv(e)$.

Proof We prove these simultaneously by inducting on the derivation of $p \xrightarrow{u} q$, $(ops, e) \rightsquigarrow (x)b$ or $fun(ops, e) \rightsquigarrow \lambda(x)b$. For example, if $p = (\mathtt{let}\ ops :: ops', e, a :: as, RS)$ and $p \xrightarrow{u} q = (ops', e, (let\ x = a\ in\ b) :: as, RS)$ where $(ops, e) \rightsquigarrow (x)b$ then by induction, $fv(b) - \{x\} \subseteq fv(e)$, and so $fv(q) = fv(as) \cup fv(e) \cup fv(let\ x = a\ in\ b) \subseteq fv(as) fv(e) \cup fv(a) = fv(p)$. □

We show that no u transition can prevent unloading, and that the unload relation $\rightsquigarrow$ is deterministic.

Lemma 3.8 *Suppose* $p \xrightarrow{u} q$. *Then for all* a, $p \rightsquigarrow a$ *if and only if* $q \rightsquigarrow a$.

Proof By determinacy of $\xrightarrow{u}$. □

Lemma 3.9 *Whenever* $p \rightsquigarrow a$ *and* $p \rightsquigarrow a'$, $a = a'$.

Proof Assume $p \rightsquigarrow a$ and $p \rightsquigarrow a'$. $p \rightsquigarrow a'$ means $p \xrightarrow{u}{}^{*} q = ([], e, [a'], [])$. By Lemma 3.8, $q \rightsquigarrow a$. But q cannot perform a u-transition (by inspection of the u-transition rules) and so $q \rightsquigarrow a'$ only. Hence $a = a'$. □

We now show that the unloading machine preserves reduction contexts under certain conditions. We use $u^{\diamondsuit}$ and $v^{\diamondsuit}$ to stand for terms which are either locations, λ abstractions or marks ($\diamondsuit$).

Lemma 3.10 *If* $(ops, e, as, RS) \xrightarrow{u} (ops', e', as', RS')$ *where* $\bullet \notin fv(e)$ *and* $as = [a_i^{\Diamond\ i \in 1..n}, \mathcal{R}, u_j^{\Diamond\ i \in 1..m}]$ *then* $\bullet \notin fv(e')$ *and* $as' = [b_i^{\Diamond\ i \in 1..n'}, \mathcal{R}', v_j^{\Diamond\ j \in 1..m'}]$ *for some* $\mathcal{R}'$, $b_i^{\Diamond}$ *and* $v_j^{\Diamond}$ *(with* $i \in 1..n', j \in 1..m'$*).*

Proof We consider each u-transition in turn.

(u **Access**) This step pushes a term onto the front of the argument stack, leaving the environment and the remainder of the stack unchanged.

(u **Object**), (u **Cur**), (u **Pushmark**), (u **Grab**) Similar to (u Access).

(u **Clone**) Here $ops = \texttt{clone} :: ops'$. If $n = 0$, so $as = [\mathcal{R}, u_1^{\Diamond}, \ldots, u_m^{\Diamond}]$ then $(ops, e, as, RS) \xrightarrow{u} (ops', e, [clone(\mathcal{R}), u_1^{\Diamond}, \ldots, u_m^{\Diamond}], RS)$ and since $clone(\mathcal{R})$ is a reduction context this satisfies the conditions of the lemma. Otherwise, in the case $n > 0$, we have that $(ops, e, as, RS) \xrightarrow{u} (ops', e, [clone(a_1^{\Diamond}), a_2^{\Diamond}, \ldots, a_n^{\Diamond}, \mathcal{R}, u_j^{\Diamond\ j \in 1..m}], RS)$.

(u **Select**) Here $ops = \texttt{select}\ \ell :: ops'$. Similarly to the ($u$ Clone) case, if $n > 0$ the conditions are easily satisfied. Otherwise, when $n = 0$, $as = [\mathcal{R}, u_1^{\Diamond}, \ldots, u_m^{\Diamond}]$ and $(ops, e, as, RS) \xrightarrow{u} (ops', e, [\mathcal{R}.\ell, u_1^{\Diamond}, \ldots, u_m^{\Diamond}], RS)$, sufficient since $\mathcal{R}.\ell$ is a reduction context.

(u **Let**) Here $ops = \texttt{let}\ ops' :: ops''$. Again, for the $n = 0$ case, by (u Let) we have $(ops', e) \rightsquigarrow (x)b$, and $as' = [let\ x = \mathcal{R}\ in\ b, u_1^{\Diamond}, \ldots, u_m^{\Diamond}]$. This is sufficient, because $(let\ x = \mathcal{R}\ in\ b)$ is a reduction context, since $\bullet \notin fv(b)$ by Lemma 3.7.

(u **Update**) Similar to (u Let).

(u **Return**) The reduction $([], e, as, (ops, E') :: RS) \xrightarrow{u} (ops, e', as, RS)$ (where $E' \rightsquigarrow e'$) leaves the argument stack unchanged, and $\bullet \notin fv(e')$ by Lemma 3.7.

(u **Function Return**) Let $p = ([\texttt{return}], e, as, (ops', E') :: RS)$ where $as = [a_i^{\Diamond\ i \in 1..n}, \mathcal{R}, u_j^{\Diamond\ j \in 1..m}]$. If $a_k^{\Diamond} = \Diamond$ and $a_i^{\Diamond} \neq \Diamond$ for $i < k$, then we have that $p \xrightarrow{u} p' = (ops', e', (a_1^{\Diamond} \ldots a_{k-1}^{\Diamond}) :: [u_{k+1}^{\Diamond}, u_{k+2}^{\Diamond}, \ldots, \mathcal{R}, u_j^{\Diamond j \in 1..m}], RS)$ where $E' \rightsquigarrow e'$. The conditions of the lemma are satisfied by p'.

Otherwise, if $a_i^{\Diamond} \neq \Diamond$ for each $i \in 1..n$, then it must be that $u_k^{\Diamond} = \Diamond$ for some $k \in 1..m$ since we are assuming (u Function Return) can be applied to p. We can pick the least such k, so that $u_i^{\Diamond} \neq \Diamond$ for $i < k$ and $u_k^{\Diamond} = \Diamond$. Now, $p \xrightarrow{u} p' = (ops', e', as' = (a_1^{\Diamond} \ldots a_n^{\Diamond} \mathcal{R} u_1^{\Diamond} \ldots u_{k-1}^{\Diamond}) :: [u_{k+1}^{\Diamond}, \ldots, u_m^{\Diamond}], RS)$. The term at the head of as' is a reduction context (since we evaluate right-to-left in applications), so the conditions of the lemma are satisfied.

(u Apply) Similar to (u Function Return) $\square$

We now show that the head of the argument stack corresponds to the part of the source expression which is currently evaluating.

Proposition 3.11 *Whenever* $(ops, e, a{::}[u_i^{\diamond\, i\in 1..n}], RS) \rightsquigarrow b$, *where* $\bullet \notin fv(e)$, *there is a reduction context,* $\mathcal{R}$, *such that* $(ops, e, a' :: [u_i^{\diamond\, i\in 1..n}], RS) \rightsquigarrow \mathcal{R}[a']$ *for any* a'.

Proof If $(ops, e, a :: [u_i^{\diamond\, i\in 1..n}], RS) \rightsquigarrow b$ then there is an a' such that $(ops, e, \bullet :: [u_i^{\diamond\, i\in 1..n}], RS) \rightsquigarrow a'$ (by Lemma 3.2).

This means $(ops, e, \bullet :: [u_i^{\diamond\, i\in 1..n}], RS) \xrightarrow{u}^{k} ([], e', [a'], [])$ for some k. Since $\bullet$ is a reduction context, applying Lemma 3.10 k times tells us that $a' = \mathcal{R}$ for some $\mathcal{R}$. Since $\bullet \notin fv(e)$, Lemma 3.6 implies $(ops, e, [a'], RS) \rightsquigarrow \mathcal{R}[a']$ (because $a' = \bullet\{\!\{a'/\bullet\}\!\}$ and $\mathcal{R}[a'] = \mathcal{R}\{\!\{a'/\bullet\}\!\}$). $\square$

We show that β transitions of the abstract machine correspond to reductions in the extended object calculus, and that τ reductions are not reflected in the source level reductions.

Lemma 3.12

(1) *If* $C \rightsquigarrow c$ *and* $C \xrightarrow{\tau} D$ *then* $D \rightsquigarrow c$.

(2) *If* $C \rightsquigarrow c$ *and* $C \xrightarrow{\beta} D$ *then there is a* d *such that* $D \rightsquigarrow d$ *and* $c \rightarrow d$.

Proof

(1) The proof for each of the τ reductions is similar. We detail only the (τ Access) case.

(τ Access) Here $C = (P, \Sigma)$, where $P = (\texttt{access}\ j :: ops, E, AS, RS)$, $E = [U_i{}^{i\in 1..n}]$, $j \in 1..n$, $C \rightsquigarrow c = (a, \sigma)$ and $C \xrightarrow{\tau} D = (Q, \Sigma)$ where $Q = (ops, E, U_j :: AS, RS)$. Now, $P \downarrow p = (\texttt{access}\ j :: ops, e, as, RS)$ where $E \rightsquigarrow e$, $e = [a_i{}^{i\in 1..n}]$, $U_i \rightsquigarrow a_i$ and $AS \rightsquigarrow as$. Similarly $Q \downarrow q = (ops, e, a_j :: as, RS)$. Since $C \rightsquigarrow (a, \sigma)$, and p is unique, $p \rightsquigarrow a$ (from the definition of (Config Unload)). By (u Access), $p \xrightarrow{u} q$, so by Lemma 3.8 and $p \rightsquigarrow a$ we have $q \rightsquigarrow a$. So $D \rightsquigarrow (a, \sigma)$ as required.

(2) We examine each rule that may derive $C \xrightarrow{\beta} D$.

(β Clone) Here $C = (P, \Sigma)$, where $P = (\mathtt{clone} :: ops, E, \iota :: AS, RS)$, and $C \xrightarrow{\beta} D = (Q, \Sigma')$ where $Q = (ops, E, \iota' :: AS, RS)$, $\Sigma' = (\iota' \mapsto \Sigma(\iota)) :: \Sigma$ and $\iota' \notin dom(\Sigma)$. We have $C \rightsquigarrow c = (a, \sigma)$ also, where $P \downarrow p$, $p = (\mathtt{clone}{::}ops, e, \iota{::}as, RS)$, $E \rightsquigarrow e$, $AS \rightsquigarrow as$, $p \rightsquigarrow a$ and $\Sigma \rightsquigarrow \sigma$. By ($u$ Clone), $p \xrightarrow{u} (ops, e, (clone(\iota)) :: as, RS)$. Hence by Lemma 3.8, $(ops, e, (clone(\iota)) :: as, RS) \rightsquigarrow a$. Therefore by Proposition 3.11, there is a reduction context $\mathcal{R}$ such that for all a', $(ops, e, a' {::} as, RS) \rightsquigarrow \mathcal{R}[a']$; by Lemma 3.9, $a = \mathcal{R}[clone(\iota)]$ and $q = (ops, e, \iota' :: as, RS) \rightsquigarrow \mathcal{R}[\iota']$. Let $\sigma' = (\iota' \mapsto \sigma(\iota)) :: \sigma$ so that $\Sigma' \rightsquigarrow \sigma'$ by (Unload Store). Let $d = (\mathcal{R}[\iota'], \sigma')$. $Q \downarrow q \rightsquigarrow \mathcal{R}[\iota']$, so $D = (Q, \Sigma') \rightsquigarrow d$. Finally, we have $c \rightarrow d$ using (Red Clone).

(β Object), (β Update) These cases work similarly.

(β Select) Here $C = (P, \Sigma)$, and $C \xrightarrow{\beta} D = (Q, \Sigma)$ where $P = (\mathtt{select}\ \ell_j :: ops, E, \iota :: AS, RS)$, $Q = (ops_j, \iota :: E_j, AS, (ops, E) :: bRS)$ and $\Sigma(\iota) = [(\ell_i, (ops_i, E_i))^{\ i \in 1..n}]$. We have $C \rightsquigarrow c = (a, \sigma)$ also, where $C \downarrow (p, \sigma)$, $p = (\mathtt{select}\ \ell_j :: ops, e, \iota :: as, RS)$, $E \rightsquigarrow e$, $AS \rightsquigarrow as$, $p \rightsquigarrow a$ and $\Sigma \rightsquigarrow \sigma$. Also, $D \downarrow (q, \sigma)$ where $q = (ops_j, \iota :: e_j, as, (ops, E) :: RS)$ and $E_j \rightsquigarrow e_j$.

By (u Select), $p \xrightarrow{u} p'$ where $p' = (ops, e, (\iota.\ell_j) :: as, RS)$. By (Unload Object), $\Sigma \rightsquigarrow \sigma$ and $\Sigma(\iota) = [(\ell_i, (ops_i, E_i))^{\ i \in 1..n}]$ we have that $(ops_j, E_j) \rightsquigarrow (y_j)a_j$ for some a_j. By (Unload Abstraction) this means $(ops_j, y_j :: E_j, [], []) \xrightarrow{u}{}^{*} ([], e', [a_j], [])$ for some e'. Hence by Lemma 3.6 we have $(ops_j, \iota :: E_j, [], []) \xrightarrow{u}{}^{*} ([], e'\{\!\{\iota/y_i\}\!\}, [a_i\{\!\{\iota/y_i\}\!\}], RS)$. By Lemma 3.3(1) we have $q \xrightarrow{u}{}^{*} q'' = ([], e'\{\!\{\iota/y_j\}\!\}, (a_j\{\!\{\iota/y_j\}\!\}) {::} as, (ops, E) {::} RS)$ and by (τ Return) $q'' \xrightarrow{u} q' = (ops, e, (a_j\{\!\{\iota/y_j\}\!\}) :: as, RS)$ (where $E \rightsquigarrow e$). By Proposition 3.11 there is a reduction context $\mathcal{R}$ such that for all a', $(ops, E, [a'], RS) \rightsquigarrow \mathcal{R}[a']$. Applying this to p' and q' we get $p' \rightsquigarrow \mathcal{R}[\iota.\ell_j]$ and $q' \rightsquigarrow \mathcal{R}[a_j\{\!\{\iota/y_j\}\!\}]$. Since $p \xrightarrow{u} p'$, Lemmas 3.9 and 3.8 give us $a = \mathcal{R}[\iota.\ell_j]$. Let $d = (\mathcal{R}[a_j\{\!\{\iota/y_j\}\!\}], \sigma)$. Then $c \rightarrow d$ and $D \rightsquigarrow d$.

(β Function Return), (β Apply), (β Let), (β Grab) These work in a similar way to (β Select). $\square$

To prove that the abstract machine simulates the object calculus semantics, we first need to prove some technical lemmas. We show that the number of τ transitions is bounded for a given state, that if a state unloads to a value then its form is restricted, and that if the abstract machine is stuck then so is its unloaded source term.

Lemma 3.13 *For all configurations C there is a D with $C \xrightarrow{\tau}^* D$ and not $D \xrightarrow{\tau}$.*

Proof Every $\xrightarrow{\tau}$ step either decreases $|RS|$ or keeps RS constant, and consumes an instruction.

The function $f : (ops, E, AC, RS) \mapsto (|RS|, |ops|)$ from states to $\mathbb{N} \times \mathbb{N}$ is such that if $C \xrightarrow{\tau} D$ then $f(D) < f(C)$ in the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$, namely $(x, y) < (x', y')$ if $x < x'$ or $x = x'$ and $y < y'$. An infinite chain $C_1 \xrightarrow{\tau} C_2 \xrightarrow{\tau} \ldots$ would give an infinite descending chain in $\mathbb{N} \times \mathbb{N}$, a contradiction since the lexicographic ordering is a well-ordering. □

Lemma 3.14

(1) *If $D \rightsquigarrow (\iota, \sigma)$ and that not $D \xrightarrow{\tau}$ then $D = (([], E, [\iota], []), \Sigma)$ for some E, Σ.*

(2) *If $D \rightsquigarrow (\lambda(x)a, \sigma)$ and not $D \xrightarrow{\tau}$ then $D = (([], E, [fun(ops, E')], []), \Sigma)$ for some E, Σ and some ops, E' such that $fun(ops, E') \rightsquigarrow \lambda(x)a$.*

Proof

(1) We have that not $D \xrightarrow{\beta}$ since by Lemma 3.12 we would have a c with $(\iota, \sigma) \to c$. Suppose $D = ((ops, E, AS, RS), \Sigma)$. By (Unload Config) we have $\Sigma \rightsquigarrow \sigma$, $E \rightsquigarrow e$, $AS \rightsquigarrow as$ and $(ops, e, as, RS) \xrightarrow{u}^* ([], e', [\iota], [])$ for some e'. First we note that $ops = []$ since otherwise (by examining cases) either D would not unload, or could make a β or a τ reduction. Similarly, $RS = []$ since otherwise (since $ops = []$) D could make a (τ Return) transition. Now, $([], e, as, [])$ cannot perform a u transition, but $([], e, as, []) \xrightarrow{u}^* ([], e', [\iota], [])$.Hence, $e = e'$ and $as = \iota$. Since $AS \rightsquigarrow as = [\iota]$ we have $AS = [\iota]$ by (Unload List Location). Hence $D = (([], E, [\iota], []), \Sigma)$.

(2) A similar argument shows that $D = (([], E, [fun(ops, E')], []), \Sigma)$ where $\Sigma \rightsquigarrow \sigma$ and $fun(ops, E') \rightsquigarrow \lambda(x)a$. □

Lemma 3.15 *If $C \rightsquigarrow c$ and there is no D such that $C \xrightarrow{\beta\tau} D$ then there is no d with $c \to d$.*

Proof Here $C \rightsquigarrow c$ means $C \downarrow (p, \sigma)$, $p \xrightarrow{u}^* ([], e, [a], [])$ and $\Sigma \rightsquigarrow \sigma$.

Firstly, if $p = ([], e, [u], [])$ so that $c = (u, \sigma)$ then c is a value, and so there is no d with $c \to d$. Otherwise, there must be a q with $p \xrightarrow{u} q$ since if $p \xrightarrow{u}^k ([], e, [a], [])$ and p is not $([], e, [a], [])$ then $k > 0$. We consider two possible inferences of $p \xrightarrow{u} q$, omitting the similar details for the other cases.

If $p = (\texttt{access}\ j :: ops, e, as, RS)$ (so $C = ((\texttt{access}\ j :: ops, E, AS, RS), \Sigma)$ where $E \rightsquigarrow e$ and $AS \rightsquigarrow as$) then $q = (ops, e, u_j :: as, RS)$ where $e = [{u_i}^{i \in 1..n}]$ and $j \in 1..n$. But now $C \xrightarrow{\tau} ((ops, E, U_j :: AS, RS), \Sigma)$ where $U_j \rightsquigarrow u_j$, contradicting the non-existence of a D with $C \xrightarrow{\beta\tau} D$.

If $p = (\texttt{select}\ \ell :: ops, e, u :: as, RS)$ (so that $C = ((\texttt{select}\ \ell :: ops, E, U :: AS, RS), \Sigma)$ where $E \rightsquigarrow e$, $U \rightsquigarrow U$ and $AS \rightsquigarrow as$). Then $q = (ops, e, (u.\ell) :: as, RS)$. By Proposition 3.11 there is a reduction context $\mathcal{R}$ such that $q \rightsquigarrow \mathcal{R}[u.\ell]$. Lemma 3.8 gives $c = (\mathcal{R}[u.\ell], \sigma)$. If $c \to d$ then we see $u = \iota$, $\sigma(\iota) = o@[\ell = \varsigma(x)a]@o'$. Hence $U = \iota$ and $\Sigma(\iota) = O@[\ell = (ops', E')]@O'$ where $(ops', E') \rightsquigarrow (x)a$. But now, $C = ((\texttt{select}\ \ell :: ops, E, \iota :: AS, RS), \Sigma)$ and $C \xrightarrow{\beta} ((ops', \iota :: E', AS, RS), \Sigma)$, giving a contradiction. □

We are now in a position to show that the abstract machine semantics simulates the semantics of the object calculus.

Lemma 3.16 *If $C \rightsquigarrow c$ and $c \to d$ then there are D, D' with $C \xrightarrow{\tau}^* D'$, $D' \xrightarrow{\beta} D$ and $D \rightsquigarrow d$.*

Proof By Lemma 3.13 we have a D' with $C \xrightarrow{\tau}^* D'$ and not $D' \xrightarrow{\tau}$. If there is no D'' with $D' \xrightarrow{\beta} D''$ then by Lemma 3.15 there is no d with $c \to d$, contradicting the assumption of this lemma. So $D' \xrightarrow{\beta} D''$ for some D''. We consider each of the β-transitions in turn.

(β Select) Here $D' = ((\texttt{select}\ \ell :: ops, E, \iota :: AS, RS), \Sigma)$ where $\Sigma(\iota) = O\,@\,[(\ell, (ops', E'))]@O'$. $D' \downarrow (p, \sigma)$ where $p = (\texttt{select}\ \ell :: ops, e, \iota :: as, RS)$, $E \rightsquigarrow e$, $AS \rightsquigarrow as$, $\Sigma \rightsquigarrow \sigma$. $p \xrightarrow{u} (ops, e, (\iota.\ell) :: as, RS)$, and by Proposition 3.11 there is a reduction context $\mathcal{R}$ such that $p \rightsquigarrow \mathcal{R}[\iota.\ell]$. Hence, $c = (\mathcal{R}[\iota.\ell], \sigma)$ and if $c \to d'$ then, since (Red Select) is the unique rule which can derive $c \to d'$ and gives a unique d', $d = d'$.

(β Let), (β Update), (β Function Return), (β Apply), (β Grab) Similar to (β Select).

(β Clone) Here $D' = (P, \Sigma) = ((\texttt{clone} :: ops, E, \iota :: AS, RS), \Sigma)$ where $\Sigma(\iota) = O$. By (u Clone), (Unload Store) and Proposition 3.11, $D' \rightsquigarrow c = (\mathcal{R}[clone(\iota)], \sigma)$ where $\sigma(\iota) = o$ and $O \rightsquigarrow o$. Now $d = (\mathcal{R}[\iota'], \sigma + (\iota' \mapsto$

$o))$ where $\iota' \notin dom(\sigma)$. By (Unload Store) $\iota' \notin dom(\Sigma)$ so by (β Clone) $D' \xrightarrow{\beta} D = ((ops, E, \iota' :: AS, RS), \Sigma + (\iota' \mapsto O))$. Invoking Proposition 3.11 again, we get $D \rightsquigarrow (\mathcal{R}[\iota'], \sigma+(\iota' \mapsto o)) = d$ as required.

(β Object) Similar to (β Clone).

□

We combine Lemmas 3.12 and 3.16 to show that the semantics of the abstract machine and that of the extended object calculus are related via the unloading relation.

Let $C \twoheadrightarrow D$ if $C \xrightarrow{\beta\tau}^{*} D$ and D is of the form $(([], E, [V], []), \Sigma)$ for some E, V and Σ. Such a D we call *terminal.* Similarly, we define $c \twoheadrightarrow d$ to mean $c \rightarrow^{*} d$ and d terminal, for c and d object calculus configurations.

Lemma 3.17

(1) *If $C \rightsquigarrow c$ and $C \twoheadrightarrow D$ then there is a d with $D \rightsquigarrow d$ and $c \twoheadrightarrow d$.*

(2) *If $C \rightsquigarrow c$ and $c \twoheadrightarrow d$ then there is a D with $D \rightsquigarrow d$ and $C \twoheadrightarrow D$.*

Proof For Part (1) we note that if $C \rightarrow^{*} D$ and $C \rightsquigarrow c$ then by repeated application of Lemma 3.12 we have that $D \rightsquigarrow d$ and $c \rightarrow^{*} d$. It remains to note that if $C \twoheadrightarrow D$ then D is a terminal configuration, and (since it unloads) it unloads to a value, so $c \twoheadrightarrow d$.

For (2), we note that $c \twoheadrightarrow d$ means $c \rightarrow^{*} d$ and $d = (v, \sigma)$. By Lemma 3.16, we have a D' with $C \rightarrow^{*} D'$ and $D' \rightsquigarrow (v, \sigma)$. By Lemma 3.13 there is a D such that $D' \xrightarrow{\tau}^{*} D$ and not $D \xrightarrow{\tau}$. By Lemma 3.12(1) we know $D \rightsquigarrow (v, \sigma) = d$, and by Lemma 3.14 we get that D is of the form $([], E, [V], []), \Sigma)$ for some E, V, Σ as required for $C \twoheadrightarrow D$.

□

We are now in a position to be able to prove the main result.

Theorem 3.18 *Whenever $[] \vdash a \Rightarrow ops$, for all d, $(a, []) \twoheadrightarrow d$ if and only if there is a D with $((ops, [], [], []), []) \twoheadrightarrow D$ and $D \rightsquigarrow d$.*

Proof Suppose we have $(a, []) \twoheadrightarrow d$. We have ops such that $[] \vdash a \Rightarrow ops$. By Proposition 3.5 we have $((ops, [], [], []), []) \rightsquigarrow (a, [])$, and we now apply part (2) of Lemma 3.17 to get that $((ops, [], [], []), []) \twoheadrightarrow D$ for some D with $D \rightsquigarrow d$.

Conversely, if we have $[] \vdash a \Rightarrow ops$ and $((ops, [], [], []), []) \twoheadrightarrow D$ for some ops and D then by Lemma 3.12 we have $((ops, [], [], []), []) \rightsquigarrow (a, [])$ and applying Lemma 3.17 part (1) we get a d with $D \rightsquigarrow d$ and $c \twoheadrightarrow d$. □

3.5 Discussion and Related Work

We have proved correct a machine based on the machine used in our implementation. The machine could be described as a ZAM (Leroy 1990) plus objects (but without some of the tail-recursion optimisations). Because of this, the proof given here can be considered as a correctness proof of a simplified ZAM, and we are sure that the proof could be scaled up to the full ZAM.

There is a large literature on proofs of interpreters based on abstract machines, such as Landin's SECD machine (Hannan 1992; Plotkin 1975; Sestoft 1994). Since no compiled machine code is involved, unloading such abstract machines is easier than unloading an abstract machine based on compiled code. The VLISP project (Guttman, Swarup, and Ramsdell 1995), using denotational semantics as a metalanguage, is the most ambitious verification to date of a compiler-based abstract machine. Other work on compilers deploys metalanguages such as calculi of explicit substitutions (Hardin, Maranget, and Pagano 1996) or process calculi (Wand 1995). Rather than introduce a metalanguage, we prove correctness of our abstract machine directly from its operational semantics. We adopted Rittri's idea (Rittri 1990) of unloading a machine state to a term via a specialised unloading machine. Our proof is simpler than Rittri's, and goes beyond it by dealing with state and objects.

There are differences, of course, between our formal model of the abstract machine and our actual implementation. One difference is that we have modelled programs as finitely branching trees, whereas in the implementation programs are tables of bytecodes indexed by a program counter. Another difference is that our model omits garbage collection, which is essential to the implementation. Therefore Theorem 3.18 only implies that the compilation strategy is correct; bugs may remain in its implementation.

4 Operational Equivalence

We now develop a theory of operational equivalence for the calculus. The standard definition of operational equivalence between terms is that of contextual equivalence (Morris 1968; Plotkin 1977): two terms are equivalent if and only if they are interchangeable in any program context without any observable difference; the observations are typically the programs' termination behaviour. Contextual equivalence is the largest congruence relation that distinguishes observably different programs. Terms are equivalent if and only if no amount of programming can tell them apart. This is a robust and reasonable definition of semantic equivalence.

Mason and Talcott (1991) have shown a useful context lemma for functional languages with state. It asserts that contextual equivalence coincides with so-called CIU equivalence. To prove two terms are CIU equivalent, one needs to show that they have identical termination behaviour when placed in the redex position in an arbitrary configuration. Although contextual equivalence and CIU equivalence are the same relation, the definition of the latter is typically easier to use in proofs.

We take CIU equivalence as our definition of operational equivalence for imperative objects and we establish some useful equivalence laws. Furthermore, we show that operational equivalence is a congruence, allowing compositional equational reasoning and a proof that it coincides with contextual equivalence. The congruence proof is adapted from the corresponding congruence proof for a λ-calculus with references by Honsell, Mason, Smith, and Talcott (1993).

We take a modular approach to formulating CIU equivalence. In Section 4.1, we introduce experimental equivalence, an auxiliary relation on configurations. In Section 4.2, we phrase our definition of operational equivalence in terms of experimental equivalence, but prove our formulation is equivalent to the one of Mason and Talcott (1991). We derive a variety of equational laws for imperative objects in Section 4.3. Section 4.4 contains our congruence proof for operational equivalence, which we use in Section 4.5 to show that operational and contextual equivalence are the same.

4.1 Experimental Equivalence

For configurations c and c', we write $c \updownarrow c'$ to mean that either both converge or neither of them converges, that is, $c\Downarrow$ if and only if $c'\Downarrow$.

We define a family of relations on configurations, called *experimental equivalence*. Recall that w ranges over finite sets of locations. Two configurations (a, σ) and (a', σ') are experimentally equivalent at index set w, written $w \vdash (a, \sigma) \sim (a', \sigma')$, if and only if $w \vdash (a, \sigma)$, $w \vdash (a', \sigma')$ and, for all reduction contexts with $locs(\mathcal{R}) \subseteq w$ and $fv(\mathcal{R}) = \{\bullet\}$, $(\mathcal{R}[a], \sigma) \updownarrow (\mathcal{R}[a'], \sigma')$.

We may regard experimental equivalence at w as a kind of testing equivalence. Let a *w-test* be a reduction context $\mathcal{R}$ such that $locs(\mathcal{R}) \subseteq w$ and $fv(\mathcal{R}) = \{\bullet\}$. Let a configuration (a, σ) *pass a w-test*, $\mathcal{R}$, if and only if $(\mathcal{R}[a], \sigma)\Downarrow$. Then two configurations c and c' are experimentally equivalent at w if and only if $w \vdash c$, $w \vdash c'$ and they pass the same w-tests.

The index set w is a view into the configurations: the locations in the stores that $\mathcal{R}$ may directly inspect. Other locations in the stores may only be inspected indirectly.

For every index set w, experimental equivalence is an equivalence relation (reflexive, transitive and symmetric) on configurations, and it is anti-monotone in the index set w:

(∼ Refl) $\dfrac{w \vdash c}{w \vdash c \sim c}$ (∼ Trans) $\dfrac{w \vdash c \sim c'' \quad w \vdash c'' \sim c'}{w \vdash c \sim c'}$ (∼ Symm) $\dfrac{w \vdash c \sim c'}{w \vdash c' \sim c}$ (∼ Anti) $\dfrac{w' \vdash c \sim c' \quad w \subseteq w'}{w \vdash c \sim c'}$

Reduction is sound with respect to experimental equivalence:

Lemma 4.1 *If $w \vdash c$ and $c \to c'$, then $w \vdash c \sim c'$.*

Proof Suppose $w \vdash (a, \sigma)$ and $(a, \sigma) \to (a', \sigma')$. Then $w \vdash (a', \sigma')$ holds by Lemma 2.1. From Lemma 2.6(1) we get that $(\mathcal{R}[a], \sigma) \to (\mathcal{R}[a'], \sigma')$, whenever $locs(\mathcal{R}) \subseteq w$. Clearly, $(\mathcal{R}[a'], \sigma')\downarrow$ implies $(\mathcal{R}[a], \sigma)\downarrow$ because any converging reduction sequence from $(\mathcal{R}[a'], \sigma')$ extends to a converging reduction sequence from $(\mathcal{R}[a], \sigma)$. The reverse implication follows because reduction is deterministic up to structural equivalence at w, that is, by a combination of Proposition 2.3 and Lemma 2.4. We conclude $w \vdash (a, \sigma) \sim (a', \sigma')$, as required. $\Box$

Up to experimental equivalence, all that matters about a configuration is whether it converges, and if so, to which terminal configuration it converges:

Lemma 4.2 *Suppose $w \vdash c$ and $w \vdash c'$. Then $w \vdash c \sim c'$ if and only if either*

(1) *both c and c' converge, that is, there are terminal d and d' such that $c \to^* d$ and $c' \to^* d'$, and moreover $w \vdash d \sim d'$, or*

(2) *neither c nor c' converges.*

Proof For the forwards direction, suppose $c = (a, \sigma)$ and $c' = (a', \sigma')$. We proceed by considering whether or not c converges, that is, whether or not there is a terminal d with $c \to^* d$. If so, let $\mathcal{R} = \bullet$ so that $(\mathcal{R}[a], \sigma) = c$. Since $(\mathcal{R}[a], \sigma)\downarrow$, $w \vdash c \sim c'$ implies $c' = (\mathcal{R}[a'], \sigma')\downarrow$, that is, there is terminal d' with $c' \to^* d'$. Lemma 4.1 implies that $w \vdash c \sim d$ and $w \vdash c' \sim d'$. These two equivalences together with $w \vdash c \sim c'$ imply $w \vdash d \sim d'$ by (∼ Symm) and (∼ Trans). On the other hand, if c does not converge, neither does c', since $w \vdash c \sim c'$ implies that if c' converges so does c. In all, we have shown that condition (1) holds if c converges, and that condition (2) holds if c does not.

For the backwards implication, we must show that conditions (1) and (2) both imply that $w \vdash c \sim c'$. Given condition (1), Lemma 4.1 asserts that

$w \vdash c \sim d$ and $w \vdash c' \sim d'$. These two equivalences, with $w \vdash d \sim d'$, ($\sim$ Symm) and ($\sim$ Trans) imply $w \vdash c \sim c'$. Finally, condition (2) implies $w \vdash c \sim c'$ by definition of experimental equivalence and Lemma 2.6(2). $\square$

It is possible to formulate garbage collection principles for unused objects in terms of experimental equivalences. For example, we can call a location ι garbage in $(a, \sigma@[\iota \mapsto o]@\sigma')$ if $(a, \sigma@\sigma')$ is well formed without $(\iota \mapsto o)$ in the store, that is, a and $\sigma@\sigma'$ make no reference to ι. The following garbage collection law says that if ι is garbage in a configuration c, it can be garbage collected up to experimental equivalence at any w such that $w \vdash c$ and $\iota \notin w$.

Lemma 4.3 *If* $w \vdash (a, \sigma@\sigma')$ *and* $\vdash \sigma@[\iota \mapsto o]@\sigma'$ *ok, then* $w \vdash (a, \sigma@[\iota \mapsto o]@\sigma') \sim (a, \sigma@\sigma')$.

Proof For every $\mathcal{R}$ with $locs(\mathcal{R}) \subseteq w$ and $fv(\mathcal{R}) = \{\bullet\}$, ι is garbage in $(\mathcal{R}[a], \sigma@[\iota \mapsto o]@\sigma')$. Therefore $(\mathcal{R}[a], \sigma@[\iota \mapsto o]@\sigma') \updownarrow (\mathcal{R}[a], \sigma@\sigma')$ follows from the following lemma. $\square$

Lemma 4.4 *Suppose* ι garbage *in* $(a, \sigma@[\iota \mapsto o]@\sigma')$. *Then* $(a, \sigma@[\iota \mapsto o]@\sigma') \rightarrow^n (v, \sigma_n@[\iota \mapsto o_n]@\sigma'_n)$ *if and only if* $o = o_n$, $\iota \notin dom(\sigma_n@\sigma'_n)$, *and* $(a, \sigma@\sigma') \rightarrow^n (v, \sigma_n@\sigma'_n)$.

Proof By inspection of the reduction rules we see that $(a, \sigma@[\iota \mapsto o]@\sigma') \rightarrow (a_1, \sigma_1@[\iota \mapsto o_1]@\sigma'_1)$ if and only if $o = o_1$, $\iota \notin dom(\sigma_1@\sigma'_1)$, and $(a, \sigma@\sigma') \rightarrow (a_1, \sigma_1@\sigma'_1)$. Furthermore, for any such transition, ι is again garbage in $(a_1, \sigma_1@[\iota \mapsto o_1]@\sigma'_1)$. The result follows by induction on the length of the computations. $\square$

Experimental equivalence is only an auxiliary relation. Our main interest is operational equivalence for static terms which we introduce below. However, the experimental equivalence relation on configurations is useful because some facts about reduction, such as Lemmas 4.1, 4.2 and 4.3, are best expressed as equivalences between configurations.

4.2 Operational Equivalence

From experimental equivalence on configurations we derive an equivalence relation on static terms, *operational equivalence*. First, let a *substitution*, ρ, be a finite map from variables to locations; we write $\rho : \{x_1, \ldots, x_n\} \rightarrow w$ whenever $\rho = [x_i \mapsto \iota_i{}^{i \in 1..n}]$ and $\iota_i \in w$ for all $i \in 1..n$. Let $a\rho$ be the term obtained from a static term a by substituting $\rho(x)$ for x for every $x \in dom(\rho)$. (These substitutions denoted by ρ are a special case of the substitutions denoted by s in Section 2.4, in which locations are the only values.) Now, we define two static terms a and a' to be operationally equivalent, written $a \approx a'$, if and only if $dom(\sigma) \vdash (a\rho, \sigma) \sim (a'\rho, \sigma)$ holds for all well formed stores σ and substitutions $\rho : fv(a) \cup fv(a') \to dom(\sigma)$.

Operational equivalence is an equivalence relation on static terms:

$$(\approx \text{Refl})\ \frac{locs(a) = \varnothing}{a \approx a} \qquad (\approx \text{Trans})\ \frac{a \approx a'' \quad a'' \approx a'}{a \approx a'} \qquad (\approx \text{Symm})\ \frac{a \approx a'}{a' \approx a}$$

We define operational equivalence only for static terms because we want to study program equivalences that programmers can use for manipulations of program text. Also, most automatic program transformations, as may take place in compilers, deal with static program text or code. Locations are dynamic entities, created during reduction of configurations. A location only carries meaning in the context with a particular store. Therefore we only consider locations in connection with configurations and experimental equivalence. Our modular formulation of operational equivalence on static terms via experimental equivalence on configurations is often convenient for proofs: after instantiation of static terms a and a' into configurations $(a\rho, \sigma)$ and $(a'\rho, \sigma)$, one can apply the simpler theory of experimental equivalence.

The following lemma asserts that operational equivalence is Mason and Talcott's CIU equivalence: static terms a and a' are equivalent if and only if all 'closed instantiations' (variable substitutions ρ and stores σ) of all 'uses' (reduction contexts $\mathcal{R}$) either both converge or both diverge:

Lemma 4.5 *For all static terms a and a', $a \approx a'$ if and only if $(\mathcal{R}[a]\rho, \sigma) \updownarrow (\mathcal{R}[a']\rho, \sigma)$, for all static reduction contexts $\mathcal{R}$, well formed stores σ, and substitutions $\rho : fv(\mathcal{R}[a]) \cup fv(\mathcal{R}[a']) \to dom(\sigma)$.*

Proof Follows straightforwardly from the definition of $\approx$ and $\sim$. For the forward implication, we use the fact that $\mathcal{R}[a]\rho = (\mathcal{R}\rho)[a\rho]$ and $\mathcal{R}\rho$ is again a reduction context. For the reverse implication, note that any reduction context $\mathcal{R}'$ can be written on the form $\mathcal{R}\rho$, for some static reduction context $\mathcal{R}$ and substitution ρ. $\square$

An easy consequence of Lemma 4.5 is that operational equivalence is preserved by static reduction contexts:

$$(\approx \text{Cong } \mathcal{R})\ \frac{a \approx a' \quad locs(\mathcal{R}) = \varnothing}{\mathcal{R}[a] \approx \mathcal{R}[a']}$$

So equivalent terms in identical static reduction contexts are again equivalent. Conversely, identical static terms in equivalent reduction contexts are also equivalent:

Lemma 4.6 *If* $\mathcal{R}[x] \approx \mathcal{R}'[x]$ *and* $x \notin \mathit{fv}(\mathcal{R}) \cup \mathit{fv}(\mathcal{R}')$*, then* $\mathcal{R}[a] \approx \mathcal{R}'[a]$*, for all static terms* a*.*

Proof We must show

$$\mathit{dom}(\sigma) \vdash (\mathcal{R}[a]\rho, \sigma) \sim (\mathcal{R}'[a]\rho, \sigma) \tag{15}$$

whenever $\vdash \sigma\ ok$ and $\rho : \mathit{fv}(\mathcal{R}[a]) \cup \mathit{fv}(\mathcal{R}'[a]) \to \mathit{dom}(\sigma)$. Note that $\mathcal{R}[a]\rho = (\mathcal{R}\rho)[a\rho]$ and $\mathcal{R}'[a]\rho = (\mathcal{R}'\rho)[a\rho]$.

If $(a\rho, \sigma)\!\downarrow$ does not hold, then both $(\mathcal{R}[a]\rho, \sigma)\!\downarrow$ and $(\mathcal{R}'[a]\rho, \sigma)\!\downarrow$ are false, by Lemma 2.6(2), hence (15) holds by Lemma 4.2.

Otherwise assume $(a\rho, \sigma) \to^* (\iota, \sigma')$, for some ι and σ'. Then $(\mathcal{R}[a]\rho, \sigma) \to^* ((\mathcal{R}\rho)[\iota], \sigma')$ and $(\mathcal{R}'[a]\rho, \sigma) \to^* ((\mathcal{R}'\rho)[\iota], \sigma')$. Now (15) follows by repeated applications of Lemma 4.1 if

$$\mathit{dom}(\sigma) \vdash ((\mathcal{R}\rho)[\iota], \sigma') \sim ((\mathcal{R}'\rho)[\iota], \sigma') \tag{16}$$

But note that $(\mathcal{R}\rho)[\iota] = \mathcal{R}[x]\rho'$ and $(\mathcal{R}'\rho)[\iota] = \mathcal{R}'[x]\rho'$ if $\rho' = (x \mapsto \iota) :: \rho$. Therefore, by the assumption $\mathcal{R}[x] \approx \mathcal{R}'[x]$ and by definition of $\approx$, we have

$$\mathit{dom}(\sigma') \vdash (\mathcal{R}[x]\rho', \sigma') \sim (\mathcal{R}'[x]\rho', \sigma')$$

hence also (16) holds, by Lemma 2.1 and ($\sim$ Anti). $\square$

4.3 Laws of Operational Equivalence

From Lemma 4.6 and the definition of operational equivalence, combined with the laws for experimental equivalence above, it is possible to show a multitude of laws of operational equivalence for the constructs of the calculus. We show a selection of laws for let, select, and object constants, and we give an equational proof of β_v-reduction for the encoding of call-by-value functions from Section 2.

The let construct satisfies laws corresponding to those of Moggi's computational λ-calculus (Moggi 1989), here presented as in Talcott (1997).

Proposition 4.7

(1) $(\mathit{let}\ x = y\ \mathit{in}\ b) \approx b\{\!\{y/x\}\!\}$

(2) $(\mathit{let}\ x = a\ \mathit{in}\ \mathcal{R}[x]) \approx \mathcal{R}[a]$*, if* $x \notin \mathit{fv}(\mathcal{R})$

Proof (1) is immediate from definition of $\approx$ and Lemma 4.1. For (2), by Lemma 4.6 it suffices to show $(\mathit{let}\ x = x\ \mathit{in}\ \mathcal{R}[x]) \approx \mathcal{R}[x]$ which is immediate from (1). $\square$

Moggi's eta law is just Proposition 4.7(2) with $\mathcal{R} = \bullet$. To prove associativity:

$$let\ x = a\ in\ (let\ x = a'\ in\ b) \quad \approx \quad let\ x = (let\ x = a\ in\ a')\ in\ b \tag{17}$$

we first rewrite the left hand side to

$$let\ x = a\ in\ (let\ x = (let\ x = x\ in\ a')\ in\ b) \tag{18}$$

by Proposition 4.7(1), ($\approx$ Cong $\mathcal{R}$) and Lemma 4.6; and then we use Proposition 4.7(2), with $\mathcal{R} = (let\ x = (let\ x = \bullet\ in\ a')\ in\ b)$, to rewrite (18) into the right hand side of (17).

The effect of invoking a method that has just been updated is the same as running the method body of the update with the self parameter bound to the updated object.

Proposition 4.8 $(a.\ell \Leftarrow \varsigma(x)b).\ell \approx (let\ x = (a.\ell \Leftarrow \varsigma(x)b)\ in\ b)$

Proof By Lemma 4.6 it suffices to show Proposition 4.8 for some $y \notin fv(b)$ in place of a. This again holds by definition of $\approx$ and four applications of Lemma 4.1. $\square$

There are laws for object constants and their interaction with the other constructs of the calculus:

Proposition 4.9 *Suppose* $o = [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]$ *and* $j \in 1..n$.

(1) $o.\ell_j \approx (let\ x_j = o\ in\ b_j)$

(2) $(o.\ell_j \Leftarrow \varsigma(x)b) \approx [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i\ ^{i \in j+1..n}]$

(3) $clone(o) \approx o$

(4) $(let\ x = o\ in\ \mathcal{R}[clone(x)]) \approx (let\ x = o\ in\ \mathcal{R}[o])$, *if* $x \notin fv(o)$

(5) $(let\ x = o\ in\ b) \approx b$, *if* $x \notin fv(b)$

(6) $(let\ x = a\ in\ let\ y = o\ in\ b) \approx (let\ y = o\ in\ let\ x = a\ in\ b)$, *if* $x \notin fv(o)$ *and* $y \notin fv(a)$

Proof Proposition 4.8 together with (2) and ($\approx$ Cong $\mathcal{R}$) imply (1):

$$\begin{aligned} o.\ell_j &\approx (o.\ell_j \Leftarrow \varsigma(x_j)b_j).\ell_j \\ &\approx let\ x_j = (o.\ell_j \Leftarrow \varsigma(x_j)b_j)\ in\ b_j \\ &\approx let\ x_j = o\ in\ b_j \end{aligned}$$

(2) and (4) are immediate from definition of $\approx$ and a few applications of Lemma 4.1.

(4) and (5) imply (3):

$$\begin{array}{rcll} clone(o) & \approx & let\ x = o\ in\ clone(x) & \text{where } x \notin fv(o) \\ & \approx & let\ x = o\ in\ o & \\ & \approx & o & \end{array}$$

(5) is direct from definition of $\approx$, Lemma 4.1 and Lemma 4.3.

Finally, we consider (6). Suppose $\vdash \sigma\ ok$ and $\rho : fv(let\ x = a\ in\ let\ y = o\ in\ b) \to dom(\sigma)$. We must show

$$dom(\sigma) \vdash ((let\ x = a\ in\ let\ y = o\ in\ b)\rho, \sigma) \sim ((let\ y = o\ in\ let\ x = a\ in\ b)\rho, \sigma) \tag{19}$$

First observe that

$$((let\ y = o\ in\ let\ x = a\ in\ b)\rho, \sigma) \to ((let\ x = a\ in\ b)\rho', (\iota \mapsto o\rho) :: \sigma)$$

where $\rho' = (y \mapsto \iota) :: \rho$, for some $\iota \notin dom(\sigma)$. Note that $a\rho' = a\rho$ and ι is garbage in $(a\rho, (\iota \mapsto o\rho) :: \sigma)$. Therefore, by Lemma 4.4, either both $(a\rho', (\iota \mapsto o\rho) :: \sigma)$ and $(a\rho, \sigma)$ go wrong or diverge, or $(a\rho', (\iota \mapsto o\rho) :: \sigma) \to^n (\iota', \sigma_n@[\iota \mapsto o\rho]@\sigma'_n)$ and $(a\rho, \sigma) \to^n (\iota', \sigma_n@\sigma'_n)$, for some n, ι', σ_n and σ'_n. If they go wrong or diverge, (19) holds by Lemma 2.6(2) and Lemma 4.2. Otherwise, again by Lemma 2.6,

$$((let\ x = a\ in\ b)\rho', (\iota \mapsto o\rho) :: \sigma) \to^n ((let\ x = \iota'\ in\ b)\rho', \sigma_n@[\iota \mapsto o\rho]@\sigma'_n)$$

$$((let\ x = a\ in\ let\ y = o\ in\ b)\rho, \sigma) \to^n ((let\ x = \iota'\ in\ let\ y = o\ in\ b)\rho, \sigma_n@\sigma'_n)$$

Each reduces further to $(b\rho'', \sigma_n@[\iota \mapsto o\rho]@\sigma'_n)$, where $\rho'' = (x \mapsto \iota') :: \rho'$. By repeated applications of Lemma 4.1, we conclude (19). $\square$

It is also possible to give equational laws for the side effects of updating and cloning, but we omit the details. Instead, let us look at an example of equational reasoning using the laws above. Recall the encoding of call-by-value functions from Section 2:

$$\begin{array}{rcl} \lambda(x)b & \stackrel{\text{def}}{=} & [arg = \varsigma(z)z.arg, val = \varsigma(s)let\ x = s.arg\ in\ b] \\ b(a) & \stackrel{\text{def}}{=} & let\ y = a\ in\ (b.arg \Leftarrow \varsigma(z)y).val \end{array}$$

where $s, y \notin fv(b)$ and $y \neq z \notin fv(a)$. From the laws for let and for object constants, we can show that β_v-reduction is valid:

$$(\lambda(x)b)(y) \approx b\{\!\{y/x\}\!\} \tag{20}$$

Let $o = [arg = \varsigma(z)y, val = \varsigma(s)let\ x = s.arg\ in\ b]$, then

$$
\begin{array}{lll}
\multicolumn{3}{l}{(\lambda(x)b)(y)} \\
\approx & ((\lambda(x)b).arg \Leftarrow \varsigma(z)y).val & \text{by Prop. 4.7(1)} \\
\approx & o.val & \text{by Prop. 4.9(2) and } (\approx \text{Cong } \mathcal{R}) \\
\approx & let\ s = o\ in\ let\ x = s.arg\ in\ b & \text{by Prop. 4.9(1)} \\
\approx & let\ x = o.arg\ in\ b & \text{by Prop. 4.7(2)} \\
\approx & let\ x = (let\ z = o\ in\ y)\ in\ b & \text{by Prop. 4.9(1) and } (\approx \text{Cong } \mathcal{R}) \\
\approx & let\ x = y\ in\ b & \text{by Prop. 4.9(5) and } (\approx \text{Cong } \mathcal{R}) \\
\approx & b\{\!\{y/x\}\!\} & \text{by Prop. 4.7(1)}
\end{array}
$$

This example as well as the derivations of some of the laws above suggest the usefulness of equational reasoning for understanding and manipulating imperative object programs.

4.4 Congruence

The derivation of (20) used the fact that operational equivalence is preserved by reduction contexts, ($\approx$ Cong $\mathcal{R}$). More generally, in order to exercise compositional equational reasoning it is necessary that operational equivalence is preserved by arbitrary term constructs. This property can be formalised in terms of compatible refinement (Gordon 1994). Given a relation on terms $\mathcal{S}$, its *compatible refinement*, $\widehat{\mathcal{S}}$, relates terms with identical outermost syntactic constructor and with immediate subterms pairwise related by $\mathcal{S}$, as defined by the rules:

(Comp x)
$$\frac{}{x\ \widehat{\mathcal{S}}\ x}$$

(Comp Object)
$$\frac{b_i\ \mathcal{S}\ b_i' \quad \forall i \in 1..n}{[\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]\ \widehat{\mathcal{S}}\ [\ell_i = \varsigma(x_i)b_i'\ ^{i \in 1..n}]}$$

(Comp Select)
$$\frac{a\ \mathcal{S}\ a'}{a.\ell\ \widehat{\mathcal{S}}\ a'.\ell}$$

(Comp Update)
$$\frac{a\ \mathcal{S}\ a' \quad b\ \mathcal{S}\ b'}{a.\ell \Leftarrow \varsigma(x)b\ \widehat{\mathcal{S}}\ a'.\ell \Leftarrow \varsigma(x)b'}$$

(Comp Clone)
$$\frac{a\ \mathcal{S}\ a'}{clone(a)\ \widehat{\mathcal{S}}\ clone(a')}$$

(Comp Let)
$$\frac{a\ \mathcal{S}\ a' \quad b\ \mathcal{S}\ b'}{let\ x = a\ in\ b\ \widehat{\mathcal{S}}\ let\ x = a'\ in\ b'}$$

Let a relation be *compatible* if and only if it contains its compatible refinement. Let a *congruence* be a compatible equivalence relation.

Proposition 4.10 *Operational equivalence is a congruence.*

Proof Operational equivalence is an equivalence relation, so it remains to show that it is compatible, that is, $a \mathrel{\widehat{\approx}} a'$ implies $a \approx a'$. The proof is adapted from the corresponding congruence proof for a λ-calculus with references in (Honsell, Mason, Smith, and Talcott 1993). We prove $a \approx a'$ by case analysis of the derivation of $a \mathrel{\widehat{\approx}} a'$.

(Comp x) Here $a = a' = x$, for some variable x, and $a \approx a'$ holds because $\approx$ is reflexive, ($\approx$ Refl).

(Comp Clone) Here $a = \mathit{clone}(a_0)$, $a' = \mathit{clone}(a'_0)$, and $a_0 \approx a'_0$. But then $a \approx a'$ is immediate from ($\approx$ Cong $\mathcal{R}$) with $\mathcal{R} = \mathit{clone}(\bullet)$.

(Comp Select) Immediate from ($\approx$ Cong $\mathcal{R}$) as in the previous case.

(Comp Update) Here $a = a_0.\ell \Leftarrow \varsigma(x)b$, $a' = a'_0.\ell \Leftarrow \varsigma(x)b'$, $a_0 \approx a'_0$ and $b \approx b'$. By ($\approx$ Cong $\mathcal{R}$), $a_0 \approx a'_0$ implies $a_0.\ell \Leftarrow \varsigma(x)b \approx a'_0.\ell \Leftarrow \varsigma(x)b$. Because $\approx$ is transitive, ($\approx$ Trans), the result follows if $a'_0.\ell \Leftarrow \varsigma(x)b \approx a'_0.\ell \Leftarrow \varsigma(x)b'$. By Lemma 4.6, this again follows if

$$y.\ell \Leftarrow \varsigma(x)b \quad \approx \quad y.\ell \Leftarrow \varsigma(x)b'$$

for some $y \notin \mathit{fv}(b)$. Consider any σ and ρ such that $\vdash \sigma\ \mathit{ok}$ and $\rho : (\{y\} \cup \mathit{fv}(b) \cup \mathit{fv}(b') - \{x\}) \to \mathit{dom}(\sigma)$. We must show that

$$\mathit{dom}(\sigma) \vdash (\iota.\ell \Leftarrow \varsigma(x)b\rho, \sigma) \sim (\iota.\ell \Leftarrow \varsigma(x)b'\rho, \sigma)$$

where $\iota = \rho(y)$. If object $\sigma(\iota)$ has no ℓ method, both configurations are stuck and the equivalence holds by Lemma 4.2. Otherwise the equivalence follows by Lemma 4.1 if

$$\mathit{dom}(\sigma) \vdash (\iota, \sigma_1) \sim (\iota, \sigma'_1)$$

where σ_1 and σ'_1 are the updated stores obtained from σ by replacing the method at label ℓ in $\sigma(\iota)$ by methods $\ell = \varsigma(x)b\rho$ and $\ell = \varsigma(x)b'\rho$, respectively. To prove this equivalence, we must show

$$(\mathcal{R}[\iota], \sigma_1) \updownarrow (\mathcal{R}[\iota], \sigma'_1)$$

for all $\mathcal{R}$ with $\mathit{locs}(\mathcal{R}) \subseteq \mathit{dom}(\sigma)$ and $\mathit{fv}(\mathcal{R}) = \{\bullet\}$. Let relation $\mathcal{T}$ relate stores with identical domains and with objects pairwise identical or having ℓ methods $\ell = \varsigma(x)b\rho$ and $\ell = \varsigma(x)b'\rho$, respectively, and all other methods identical. In particular, $\sigma_1\ \mathcal{T}\ \sigma'_1$. We shall argue that

$$(a, \sigma) \updownarrow (a, \sigma') \quad \text{for all } a,\ \sigma \text{ and } \sigma' \text{ such that } \sigma\ \mathcal{T}\ \sigma'$$

Suppose $(a,\sigma)\downarrow$, that is, there exist n and a terminal configuration d such that $(a,\sigma) \rightarrow^n d$, then we show $(a,\sigma')\downarrow$ by induction on n:

If $n = 0$, (a,σ) is a terminal configuration, that is, a is a value, and so is (a,σ').

Otherwise there is (a_1,σ_1) such that $(a,\sigma) \rightarrow (a_1,\sigma_1) \rightarrow^{n-1} d$. By inspection of the reduction rules we see that $(a,\sigma') \rightarrow (a_1,\sigma_1')$ with $\sigma_1 \ \mathcal{T} \ \sigma_1'$, unless a is of the form $a = \mathcal{R}[\iota.\ell]$ where $\sigma(\iota)$ and $\sigma'(\iota)$ have methods $\ell = \varsigma(x)b\rho$ and $\ell = \varsigma(x)b'\rho$, respectively. In that case $(a_1,\sigma_1) = (\mathcal{R}[b\rho'],\sigma)$ and $(a,\sigma') \rightarrow (\mathcal{R}[b'\rho'],\sigma')$ where $\rho' = (x \mapsto \iota) :: \rho$. Since $(\mathcal{R}[b\rho'],\sigma) \rightarrow^{n-1} d$ in one less step than $(a,\sigma) \rightarrow^n d$, we get $(\mathcal{R}[b\rho'],\sigma')\downarrow$ by induction hypothesis. Moreover, $b \approx b'$ implies $dom(\sigma') \vdash (b\rho',\sigma') \sim (b'\rho',\sigma')$. Hence $(\mathcal{R}[b\rho'],\sigma') \updownarrow (\mathcal{R}[b'\rho'],\sigma')$ and we obtain $(\mathcal{R}[b'\rho'],\sigma')\downarrow$ and $(a,\sigma')\downarrow$, as required.

This completes the induction on n and we conclude $(a,\sigma)\downarrow$ implies $(a,\sigma')\downarrow$. The reverse implication is symmetrical. Therefore $(a,\sigma) \updownarrow (a,\sigma')$, as required.

(Comp Object) Follows from case (Comp Update) by repeated applications of Proposition 4.9(2).

(Comp Let) Here $a = (let\ x = a_0\ in\ b)$, $a' = (let\ x = a_0'\ in\ b)$, $a_0 \approx a_0'$ and $b \approx b'$. Firstly, $a_0 \approx a_0'$ implies $(let\ x = a_0\ in\ b) \approx (let\ x = a_0'\ in\ b)$, by ($\approx$ Cong $\mathcal{R}$). Next, $b \approx b'$ implies $(let\ x = x\ in\ b) \approx (let\ x = x\ in\ b')$ and $(let\ x = a_0'\ in\ b) \approx (let\ x = a_0'\ in\ b')$, by Proposition 4.7(1) and Lemma 4.6. Finally, $a \approx a'$ because $\approx$ is transitive, ($\approx$ Trans). □

4.5 Contextual Equivalence

We call a relation $\mathcal{S}$ on static terms *adequate* if and only if $a\ \mathcal{S}\ a'$ implies $(a,[]) \updownarrow (a',[])$, for all closed terms a and a'.

Proposition 4.11 *Operational equivalence is adequate.*

Proof Immediate from the definition of operational and experimental equivalence, by taking the empty substitution, empty store, and empty reduction context. □

Proposition 4.12 *Operational equivalence is the largest compatible and adequate relation on static terms.*

Proof We must show that any compatible and adequate relation $\mathcal{S}$ is included in $\approx$.

Suppose $a \mathrel{\mathcal{S}} a'$. By appeal to Lemma 4.5, $a \approx a'$ holds if

$$(\mathcal{R}[a]\rho, \sigma) \updownarrow (\mathcal{R}[a']\rho, \sigma) \tag{21}$$

for any given static reduction context $\mathcal{R}$ and any σ and ρ such that $\vdash \sigma \ ok$ and $\rho : fv(\mathcal{R}[a]) \cup fv(\mathcal{R}[a']) \to dom(\sigma)$.

Suppose $\sigma = [\iota_i \mapsto o_i \ ^{i \in 1..n}]$, $o_i = [\ell_{ij} = \varsigma(x_{ij})a_{ij} \ ^{j \in 1..q_i}]$, and $\rho = [x_h \mapsto \iota_{i_h} \ ^{h \in 1..m}]$ with $\{i_1 \ldots i_m\} \subseteq \{1 \ldots n\}$. Then let $\omega_i = [\ell_{ij} = \varsigma(x)x.\ell_{ij} \ ^{j \in 1..q_i}]$, pick n distinct variables $z_1 \ldots z_n$, and let b_{ij} be obtained from a_{ij} by replacing every occurrence of location ι_k by variable z_k for $k \in 1..n$, for all $j \in 1..q_i$ and $i \in 1..n$. Let

$$\begin{array}{rcl} b & = & \mathit{let}\ z_1 = \omega_1\ \mathit{in}\ \ldots \mathit{let}\ z_n = \omega_n\ \mathit{in} \\ & & \quad (...(z_1.\ell_{11} \Leftarrow \varsigma(x_{11})b_{11})...).\ell_{1q_1} \Leftarrow \varsigma(x_{1q_1})b_{1q_1}; \\ & & \quad \vdots \\ & & \quad (...(z_n.\ell_{n1} \Leftarrow \varsigma(x_{n1})b_{n1})...).\ell_{nq_n} \Leftarrow \varsigma(x_{nq_n})b_{nq_n}; \\ & & \quad \mathit{let}\ x_1 = z_{i_1}\ \mathit{in}\ \ldots \mathit{let}\ x_m = z_{i_m}\ \mathit{in}\ \mathcal{R}[a] \end{array}$$

and let b' be the same as b but with a' in place of a (notation $a; b$ abbreviates $\mathit{let}\ x = a\ \mathit{in}\ b$ where $x \notin fv(b)$). Then $b \mathrel{\mathcal{S}} b'$ holds, since $a \mathrel{\mathcal{S}} a'$ and $\mathcal{S}$ is compatible, and therefore $(b, []) \updownarrow (b', [])$, since $\mathcal{S}$ is adequate. One can check that $(b, []) \to^* (\mathcal{R}[a]\rho, \sigma)$ and $(b', []) \to^* (\mathcal{R}[a']\rho, \sigma)$. By determinacy of reduction, as in the proof of Lemma 4.1, it follows that $(a\rho, \sigma) \updownarrow (b, [])$ and $(b', []) \updownarrow (a'\rho, \sigma)$. Finally, we conclude (21), as required, because the relation $\updownarrow$ is transitive. □

Clearly, operational equivalence is also the largest adequate congruence on static terms. It follows that it coincides with Morris-style contextual equivalence, sometimes known as observational congruence (Meyer and Cosmadakis 1988), where we take convergence of programs as our means of observation. Instead of the usual definition of contextual equivalence in terms of variable capturing contexts, one can equivalently define it as the relation between static terms which are related by a compatible and adequate relation; more concretely, for any two terms a and a', let $\{(a, a')\}^c$ be the least compatible relation that relates them, defined inductively by the rules:

(Ctx a a')
$$\frac{}{a \ \{(a, a')\}^c \ a'}$$

(Ctx Comp)
$$\frac{b \ \widehat{\{(a, a')\}^c} \ b'}{b \ \{(a, a')\}^c \ b'}$$

Then a and a' are contextually equivalent if and only if $\{(a, a')\}^c$ is adequate. The coincidence between operational and contextual equivalence reads as follows:

Theorem 4.13 *$a \approx a'$ if and only if $\{(a, a')\}^c$ is adequate.*

Proof The 'if' direction is immediate from the previous proposition because $a\ \{(a, a')\}^c\ a'$ and $\{(a, a')\}^c$ is compatible and adequate. Conversely, $\{(a, a')\}^c$ is contained in $\approx$, by induction on the definition of $\{(a, a')\}^c$, since $\approx$ is closed under (Ctx a a') and (Ctx Comp) by the assumption $a \approx a'$ and by ($\approx$ Comp). Therefore $\{(a, a')\}^c$ is adequate since $\approx$ is adequate. $\square$

4.6 Discussion and Related Work

The definitions of experimental equivalence and operational equivalence are formulated in terms of reduction contexts, stores and substitutions. That makes it easy to relate experimental and operational equivalence to the substitution-based operational semantics in equivalence proofs. In contrast, the definition of contextual equivalence is robust and abstract because it is not dependent on details of the operational semantics: it only refers to static terms and adequacy (convergence). Theorems 2.7, 2.14, and 3.18 imply that adequacy can equivalently be defined on basis of any of the three operational semantics of Section 2 or the abstract machine of Section 3. Furthermore, the definition of adequacy is unaffected by the choice of store model for the operational semantics.

The object store model is well-suited for operational reasoning because it makes clear that method updates are not shared between different labels and different objects. For example, the following law follows easily from the definition of operational equivalence and Lemma 4.1.

$$(y.\ell \Leftarrow \varsigma(x)b); (z.\ell' \Leftarrow \varsigma(x')b'); a \quad \approx \quad (z.\ell' \Leftarrow \varsigma(x')b'); (y.\ell \Leftarrow \varsigma(x)b); a \qquad (22)$$

where $\ell \neq \ell'$ and notation $a; b$ abbreviates *let* $x = a$ *in* b where $x \notin fv(b)$.

In the method store model of Abadi and Cardelli (1996), object values are of the form $[\ell_i \mapsto \iota_i{}^{i \in 1..n}]$ and stores map locations to methods. A static term would be instantiated to a configuration by applying a substitution of free variables to object values and by pairing the resulting term with an associated method store. The definition of CIU equivalence would have to constrain the object values and method store used in instantiations: the resulting configuration would need to be such that different occurrences of object values do not share methods unless the occurrences are identical. For suppose we instantiate (22) with $y = [\ell \mapsto \iota]$, $z = [\ell' \mapsto \iota]$, and method store $[\iota \mapsto \varsigma(x)[]]$, and suppose $b = x$, $b' = x'.\ell'$, and $a = z.\ell'$. Then the left hand side of (22) diverges, whereas the right hand side converges to $([\ell' \mapsto \iota], [\iota \mapsto \varsigma(x)x])$.

On the other hand, one advantage of the method store model is that it makes it easy to verify that different copies of the empty object are equivalent, for instance,

$$let\ x = []\ in\ [\ell = \varsigma(s)x] \quad \approx \quad [\ell = \varsigma(s)[]] \tag{23}$$

is an instance of Proposition 4.7(1) because [] is a value. In our object store model, the proof of (23) becomes somewhat involved and requires a tedious argument analogous to that of Proposition 4.3.

The congruence proof we have presented, based on that of Honsell, Mason, Smith, and Talcott (1993), is quite simple, considering that the imperative object calculus is a higher-order, state-based language. Alternatively, it is possible to adapt Howe's general method for proving congruence of simulation orderings (Howe 1996) to CIU equivalence; see Gordon (1997) for an example of this for the stateless object calculus of Abadi and Cardelli (1996). Talcott (1997) presents another proof method based on a notion of uniform computation. These proof methods scale up more smoothly when, for example, functions are added to the calculus, but for the core calculus our direct approach is simpler.

Earlier work on operational equivalence for object calculi has been concerned with stateless objects: Gordon and Rees (1996) and Gordon (1997) characterise contextual equivalence exactly via forms of bisimilarity induced by the primitive operational semantics of objects; Sangiorgi (1996) and Hüttel and Kleist (1996) induce approximations to contextual equivalence via translations of objects into the π-calculus; Andersen, Pedersen, Hüttel, and Kleist (1997) characterise the bisimilarity of Gordon and Rees via a modal logic.

The main influence on this section has been the literature on operational theories for functional languages with state. CIU equivalence was introduced by Mason and Talcott (1991) and has been the topic of much research; see Talcott (1997) for an overview of this work as well as a more general presentation of the theory. Functional languages with state accommodate imperative object-oriented programming styles; see for example Abelson and Sussman (1985). Operational equivalences of imperative objects in this style have been studied using CIU equivalence by Mason and Talcott (1991, 1992, 1994). But program equivalences for imperative object-oriented languages do not seem to have received much study so far. Our results are a first step and indicate an interesting algebra of imperative objects. There are many subtleties with regard to operational equivalences for state-based languages, as have been highlighted by Meyer and Sieber (1988). Operational methods hold some promise in this direction (Honsell, Mason, Smith, and Talcott 1993; Pitts 1997). These subtleties are highly relevant for objects and the associated operational methods should be interesting to study for objects too, but we have not explored these issues here in any depth.

Several authors have studied operational equivalences for languages with concurrent objects (Agha, Mason, Smith, and Talcott 1997; Jones 1995; Walker 1995; Sangiorgi 1997), but the technique of CIU equivalence has yet to be recast in a concurrent setting.

5 Example: Static Resolution of Labels

In Section 3 we showed how to compile the imperative object calculus to an abstract machine that represents objects as finite lists of labels paired with method closures. In each pair, the first component is the label, and the second component is the method closure. A frequent operation is to *resolve a method label*, that is, to compute the offset of the method with that label from the beginning of the list. This operation is needed to implement both method select and method update. In general, resolution of method labels needs to be carried out dynamically since one cannot in general compute statically the object to which a select or an update will apply. However, when the select or update is performed on a newly created object, or to self, it is possible to resolve method labels statically. The purpose of this section is to exercise our framework by presenting an algorithm for statically resolving method labels in these situations, and proving it correct.

We begin in Section 5.1 by extending our calculus to allow method selects and method updates with respect to integer offsets as well as labels. We present the optimisation algorithm in Section 5.2, and prove its correctness in Section 5.3. We discuss related work in Section 5.4.

5.1 Integer Offsets

To represent our intermediate language, we begin by extending the syntax of terms so that selects and updates may be performed on (positive) integer offsets, i or j.

$a, b ::= \ldots \mid a.j \mid a.j \Leftarrow \varsigma(x)b$ terms, $0 < j$

As before, we say that a term, a, of this extended language is a *static term* if and only if $locs(a) = \varnothing$.

The intention is that at runtime, a resolved select $o.j$ proceeds by running the jth method of object o. If the jth method of object o has label ℓ, this will have the same effect as $o.\ell$. Similarly, an update $o.j \Leftarrow \varsigma(x)b$ proceeds by updating the jth method of object o with method $\varsigma(x)b$. If the jth method of object o has label ℓ, this will have the same effect as $o.\ell \Leftarrow \varsigma(x)b$.

To make this precise, the operational semantics of Section 2 and the abstract machine and compiler of Section 3 may easily be extended with integer offsets. We suppress all the details apart from the following.

We extend the reduction contexts of Section 2.2 as follows:

$\mathcal{R} ::= \ldots \mid \mathcal{R}.j \mid \mathcal{R}.j \Leftarrow \varsigma(x)b$ reduction context

We extend the small-step substitution-based semantics of Section 2.2 and the big-step substitution-based semantics of Section 2.3 with these rules:

(Red Offset Select)

$$\frac{\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}] \quad j \in 1..n}{(\mathcal{R}[\iota.j], \sigma) \to (\mathcal{R}[b_j \{\!\{\iota/x_j\}\!\}], \sigma)}$$

(Red Offset Update)

$$\frac{\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}] \quad j \in 1..n \qquad \sigma' = \sigma + (\iota \mapsto [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i\ ^{i \in j+1..n}])}{(\mathcal{R}[\iota.j \Leftarrow \varsigma(x)b], \sigma) \to (\mathcal{R}[\iota], \sigma')}$$

(Subst Offset Select)

$$\frac{(a, \sigma_0) \Downarrow (\iota, \sigma_1) \quad \sigma_1(\iota) = [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}] \quad j \in 1..n \quad (b_j \{\!\{\iota/x_j\}\!\}, \sigma_1) \Downarrow (v, \sigma_2)}{(a.j, \sigma_0) \Downarrow (v, \sigma_2)}$$

(Subst Offset Update)

$$\frac{(a, \sigma_0) \Downarrow (\iota, \sigma_1) \quad \sigma_1(\iota) = [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}] \quad j \in 1..n \qquad \sigma_2 = \sigma_1 + (\iota \mapsto [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i\ ^{i \in j+1..n}])}{(a.j \Leftarrow \varsigma(x)b, \sigma_0) \Downarrow (\iota, \sigma_2)}$$

All the results proved in Sections 2 and 3 remain true for this extended language.

The reduction contexts used in the definition of experimental equivalence now include include selects and updates with integer offsets. By enriching the syntax with integer offsets we make both experimental equivalence and operational equivalence finer grained. For instance, in the original language the order of methods in an object may be permuted without affecting operational equivalence. For example, if $a = [\ell_1 = [], \ell_2 = \varsigma(s)s.\ell_2]$ and $b = [\ell_2 = \varsigma(s)s.\ell_2, \ell_1 = []]$, then $a \approx b$. But this equation fails in the presence of reduction contexts with integer offsets, since, for instance, $(a.1, [])$ converges but $(b.1, [])$ diverges. Although the equivalences are finer grained, all the results proved in Section 4 hold for the extended calculus.

5.2 A Static Resolution Algorithm

We need the following definitions to express the static resolution algorithm.

$A ::= [\ell_i\ {}^{i \in 1..n}]$	layout type, ℓ_i distinct
$\mathit{layout}(o) = [\ell_i\ {}^{i \in 1..n}]$	where $o = [\ell_i = \varsigma(x_i)b_i\ {}^{i \in 1..n}]$
$E ::= [x_i \mapsto A_i\ {}^{i \in 1..n}]$	environment, x_i distinct

The algorithm computes a layout type, A, for each term it encounters. If the layout type A is $[\ell_i\ {}^{i \in 1..n}]$, with $n > 0$, the term must evaluate to an object o with $\mathit{layout}(o) = A$. On the other hand, if the layout type A is $[]$, nothing has been determined about the layout of the object to which the term will evaluate. An environment E is a finite map that associates layout types to the free variables of a term.

We express the algorithm as the following recursive routine $\mathit{resolve}(E, a)$, that takes an environment E and a static term a with $\mathit{fv}(a) \subseteq \mathit{dom}(E)$, and produces a pair (a', A), where static term a' is the residue of a after resolution of a number of labels to integer offsets, and A is the layout type of both a and a'. We use p to range over both labels and integer offsets.

$\mathit{resolve}(E, x) \stackrel{\text{def}}{=} (x, E(x)) \quad$ where $x \in \mathit{dom}(E)$

$\mathit{resolve}(E, [\ell_i = \varsigma(x_i)a_i\ {}^{i \in 1..n}]) \stackrel{\text{def}}{=} ([\ell_i = \varsigma(x_i)a'_i\ {}^{i \in 1..n}], A)$
  where $A = [\ell_i\ {}^{i \in 1..n}]$
  and $(a'_i, B_i) = \mathit{resolve}((x_i \mapsto A) :: E, a_i)$, $x_i \notin \mathit{dom}(E)$, for each $i \in 1..n$

$$\mathit{resolve}(E, a.p) \stackrel{\text{def}}{=} \begin{cases} (a'.j, []) & \text{if } j \in 1..n \text{ and } p = \ell_j \\ (a'.p, []) & \text{otherwise} \end{cases}$$
  where $(a', [\ell_i\ {}^{i \in 1..n}]) = \mathit{resolve}(E, a)$

$$\mathit{resolve}(E, a.p \Leftarrow \varsigma(x)b) \stackrel{\text{def}}{=} \begin{cases} (a'.j \Leftarrow \varsigma(x)b', A) & \text{if } j \in 1..n \text{ and } p = \ell_j \\ (a'.p \Leftarrow \varsigma(x)b', A) & \text{otherwise} \end{cases}$$
  where $(a', A) = \mathit{resolve}(E, a)$, $A = [\ell_i\ {}^{i \in 1..n}]$
  and $(b', B) = \mathit{resolve}((x \mapsto A) :: E, b)$, $x \notin \mathit{dom}(E)$

$resolve(E, clone(a)) \stackrel{\text{def}}{=} (clone(a'), A)$ where $(a', A) = resolve(E, a)$
$resolve(E, let\ x = a\ in\ b) \stackrel{\text{def}}{=} (let\ x = a'\ in\ b', B)$
where $(a', A) = resolve(E, a)$
and $(b', B) = resolve((x \mapsto A) :: E, b)$, $x \notin dom(E)$

To illustrate the algorithm in action, suppose that *false* is the object:

$$[val = \varsigma(s)s.f\!f, tt = \varsigma(s)[], f\!f = \varsigma(s)[]]$$

Then $resolve([], false)$ returns the following:

$$([val = \varsigma(s)s.3, tt = \varsigma(s)[], f\!f = \varsigma(s)[]], [val, tt, f\!f])$$

The method select $s.f\!f$ has been statically resolved to $s.3$. The layout type $[val, tt, f\!f]$ asserts that *false* will evaluate to an object with this layout.

5.3 Verification of the Algorithm

To allow proofs by induction on derivations, we begin by representing the algorithm by an inductively defined relation, $\leftrightarrow$. We need an auxiliary notion of a *store type*, a finite map sending locations to layout types:

$\Sigma ::= [\iota_i \mapsto A_i\ ^{i \in 1..n}]$	store type, ι_i distinct

By the following rules, we define a *resolution* relation on terms, $(E, \Sigma) \vdash a \leftrightarrow a' : A$, intended to mean that in environment E and store type Σ, and at layout type A, term a may be resolved to term a' by turning some of the labels in a into integer offsets in b.

(Layout x)
$$\frac{x \in dom(E)}{(E, \Sigma) \vdash x \leftrightarrow x : E(x)}$$

(Layout ι)
$$\frac{\iota \in dom(\Sigma)}{(E, \Sigma) \vdash \iota \leftrightarrow \iota : \Sigma(\iota)}$$

(Layout Object) (where $B = [\ell_i\ ^{i \in 1..n}]$ and $x_i \notin dom(E)$)
$$\frac{((x_i \mapsto B) :: E, \Sigma) \vdash a_i \leftrightarrow a'_i : A_i \qquad \forall i \in 1..n}{(E, \Sigma) \vdash [\ell_i = \varsigma(x_i)a_i\ ^{i \in 1..n}] \leftrightarrow [\ell_i = \varsigma(x_i)a'_i\ ^{i \in 1..n}] : B}$$

(Layout Select 1)
$$\frac{(E, \Sigma) \vdash a \leftrightarrow a' : A}{(E, \Sigma) \vdash a.\ell \leftrightarrow a'.\ell : []}$$

(Layout Select 2)
$$\frac{(E, \Sigma) \vdash a \leftrightarrow a' : A}{(E, \Sigma) \vdash a.j \leftrightarrow a'.j : []}$$

(Layout Select 3) (where $j \in 1..n$ and $\ell_j = \ell$)

$$\frac{(E,\Sigma) \vdash a \leftrightarrow a' : [\ell_i{}^{i \in 1..n}]}{(E,\Sigma) \vdash a.\ell \leftrightarrow a'.j : []}$$

(Layout Update 1) (where $x \notin dom(E)$)

$$\frac{(E,\Sigma) \vdash a \leftrightarrow a' : A \qquad ((x \mapsto A) :: E, \Sigma) \vdash b \leftrightarrow b' : B}{(E,\Sigma) \vdash a.\ell \Leftarrow \varsigma(x)b \leftrightarrow a'.\ell \Leftarrow \varsigma(x)b' : A}$$

(Layout Update 2) (where $x \notin dom(E)$)

$$\frac{(E,\Sigma) \vdash a \leftrightarrow a' : A \qquad ((x \mapsto A) :: E, \Sigma) \vdash b \leftrightarrow b' : B}{(E,\Sigma) \vdash a.j \Leftarrow \varsigma(x)b \leftrightarrow a'.j \Leftarrow \varsigma(x)b' : A}$$

(Layout Update 3) (where $x \notin dom(E)$, $j \in 1..n$ and $\ell_j = \ell$)

$$\frac{(E,\Sigma) \vdash a \leftrightarrow a' : A \qquad ((x \mapsto A) :: E, \Sigma) \vdash b \leftrightarrow b' : [\ell_i{}^{i \in 1..n}]}{(E,\Sigma) \vdash a.\ell \Leftarrow \varsigma(x)b \leftrightarrow a'.j \Leftarrow \varsigma(x)b' : A}$$

(Layout Clone)

$$\frac{(E,\Sigma) \vdash a \leftrightarrow a' : A}{(E,\Sigma) \vdash clone(a) \leftrightarrow clone(a') : A}$$

(Layout Let) (where $x \notin dom(E)$)

$$\frac{(E,\Sigma) \vdash a \leftrightarrow a' : A \qquad ((x \mapsto A) :: E, \Sigma) \vdash b \leftrightarrow b' : B}{(E,\Sigma) \vdash let\ x = a\ in\ b \leftrightarrow let\ x = a'\ in\ b' : B}$$

We need the (Layout ι) rule and store types so that the resolution relation is defined on arbitrary terms. Even though the $resolve(E, a)$ routine takes a static term a as its input, we cannot simply define the resolution relation on static terms. If we did so, we would not be able to prove Proposition 5.3, which relates resolution and evaluation, since terms containing locations may arise from evaluation of static terms.

This resolution relation on terms includes all the possible outcomes of running the algorithm:

Lemma 5.1 *Suppose that a is a static term and E is an environment with $fv(a) \subseteq dom(E)$. If routine $resolve(E, a)$ returns (a', A), then the judgment $(E, []) \vdash a \leftrightarrow a' : A$ is derivable.*

Proof By induction on the number of recursive calls made by the routine $resolve(E, a)$, using all the rules but (Layout ι). □

For example, via (Layout Object), (Layout x) and (Layout Select 3) we may derive:

$$([], []) \vdash \mathit{false} \leftrightarrow [\mathit{val} = \varsigma(s)s.3, \mathit{tt} = \varsigma(s)[], \mathit{ff} = \varsigma(s)[]] : [\mathit{val}, \mathit{tt}, \mathit{ff}]$$

The type $[\mathit{val}, \mathit{tt}, \mathit{ff}]$ asserts that both the terms in the judgment are expected to yield objects with this layout.

Given the previous judgment and (Layout Select 3) we may derive:

$$([], []) \vdash \mathit{false}.\mathit{val} \leftrightarrow [\mathit{val} = \varsigma(s)s.3, \mathit{tt} = \varsigma(s)[], \mathit{ff} = \varsigma(s)[]].1 : []$$

In this case, the empty layout type $[]$ simply asserts that nothing is known about the layout of the objects returned by the two terms (except that they will have the same layout).

We will make precise the connection between evaluation and resolution in Proposition 5.3. Since evaluation is defined on configurations, to state the proposition we first need need to extend the resolution relation to stores and configurations. By the following rules, we define a resolution relation, $\vdash \sigma \leftrightarrow \sigma' : \Sigma$, on store pairs, and another, $\vdash c \leftrightarrow c' : (A, \Sigma)$, on configuration pairs:

(Layout Store) (where $dom(\Sigma) = dom(\sigma) = dom(\sigma')$)

$$\frac{\Sigma(\iota) = \mathit{layout}(\sigma(\iota)) = \mathit{layout}(\sigma'(\iota)) \quad ([], \Sigma) \vdash \sigma(\iota) \leftrightarrow \sigma'(\iota) : \Sigma(\iota) \qquad \forall \iota \in dom(\Sigma)}{\vdash \sigma \leftrightarrow \sigma' : \Sigma}$$

(Layout Config)

$$\frac{([], \Sigma) \vdash a \leftrightarrow a' : A \qquad \Sigma \vdash \sigma \leftrightarrow \sigma'}{\vdash (a, \sigma) \leftrightarrow (a', \sigma') : (A, \Sigma)}$$

For a simple example, let o_1 and o_2 be the objects $[\ell = \varsigma(x)[]]$ and $[\ell = \varsigma(x)o_1]$ respectively. Let c be the configuration $(o_2.\ell, [])$. Using the rules above, we may derive $\vdash c \leftrightarrow c : ([], [])$. Given these rules, we cannot derive a layout type other than $[]$ for $o_2.\ell$.

To see the effect of evaluation on the layout type of this configuration, let σ be the store $[\iota_2 \mapsto o_1, \iota_1 \mapsto o_2]$. Using the evaluation rules from Section 2.3, we may derive $c \Downarrow (\iota_2, \sigma)$. (Object o_2 is allocated at location ι_1, and then object o_1 is allocated at location ι_2. The object $[]$ embedded in o_1 is not allocated.) Moreover, using the rules above, we may derive:

$$\vdash (\iota_2, \sigma_2) \leftrightarrow (\iota_2, \sigma_2) : ([\ell], [\iota_1 \mapsto [\ell], \iota_2 \mapsto [\ell]])$$

This example shows that, as one might expect, evaluation increases the accuracy of the layout types derivable for a configuration. In seeking to verify the *resolve* routine, we introduced the resolution relation because it includes all the results of running *resolve*, Lemma 5.1, but also because we can prove that resolution is preserved by evaluation, Proposition 5.3. We first need the following substitution lemma.

Lemma 5.2 *$(E'@[x \mapsto A]@E'', \Sigma) \vdash a \leftrightarrow a' : B$, $\iota \in dom(\Sigma)$ and $\Sigma(\iota) = A$ imply $(E'@E'', \Sigma) \vdash a\{\!\{\iota/x\}\!\} \leftrightarrow a'\{\!\{\iota/x\}\!\} : B$.*

Proof A routine induction on the derivation of the judgment $(E'@[x \mapsto A]@E'', \Sigma) \vdash a \leftrightarrow a' : B$. $\square$

If Σ and Σ' are store types, let $\Sigma \leq \Sigma'$ if and only if $dom(\Sigma) \subseteq dom(\Sigma')$ and $\Sigma(\iota) = \Sigma'(\iota)$ for each $\iota \in dom(\Sigma)$.

Proposition 5.3 *Suppose that $\vdash c \leftrightarrow c' : (A, \Sigma)$.*

(1) *Whenever $c \Downarrow d$ there are d', A' and Σ' such that $c' \Downarrow d'$, $\vdash d \leftrightarrow d' : (A', \Sigma')$ and $\Sigma \leq \Sigma'$. Moreover, $A \neq []$ implies $A = A'$.*

(2) *Whenever $c' \Downarrow d'$ there are d, A' and Σ' such that $c \Downarrow d$ and $\vdash d \leftrightarrow d' : (A', \Sigma')$ and $\Sigma \leq \Sigma'$. Moreover, $A \neq []$ implies $A = A'$.*

Proof We shall prove part (1); part (2) follows by an almost symmetric argument. We show that $c \Downarrow d$ and $\vdash c \leftrightarrow c' : (A, \Sigma)$ imply there is d', A' and Σ' such that $c' \Downarrow d'$ and $\vdash d \leftrightarrow d' : (A, \Sigma)$, and $A \neq []$ implies $A = A'$, by induction on the derivation of $c \Downarrow d$.

We show the case for (Subst Select).

(Subst Select) In this case $c = (a.\ell_j, \sigma_0)$, $(a, \sigma_0) \Downarrow (\iota, \sigma_1)$, $\sigma_1(\iota) = [\ell_i = \varsigma(x_i)b_i\,^{i \in 1..n}]$, $j \in 1..n$ and $(b_j\{\!\{\iota/x_j\}\!\}, \sigma_1) \Downarrow d$. Only (Layout Config) can derive $\vdash (a.\ell_j, \sigma_0) \leftrightarrow c' : (A, \Sigma)$, so c' must take the form (b', σ_0') with $([], \Sigma) \vdash a.\ell_j \leftrightarrow b' : A$ and $\vdash \sigma_0 \leftrightarrow \sigma_0' : \Sigma$. Either (Layout Select 1), (Layout Select 2) or (Layout Select 3) can derive $([], \Sigma) \vdash a.\ell_j \leftrightarrow b' : A$. We shall consider the latter case:

(Layout Select 3) Here $b' = a'.j$ and $A = []$, with $([], \Sigma) \vdash a \leftrightarrow a' : B$, $j \in 1..m$ and $\ell_j' = \ell$ where $B = [\ell_i'\,^{i \in 1..m}]$. Since $j \in 1..m$, $B \neq []$. By (Layout Config), $\vdash \sigma_0 \leftrightarrow \sigma_0' : \Sigma$ and $([], \Sigma) \vdash a \leftrightarrow a' : B$ imply $\vdash (a, \sigma_0) \leftrightarrow (a', \sigma_0') : (B, \Sigma)$. Hence, by induction hypothesis, $(a, \sigma_0) \Downarrow (\iota, \sigma_1)$ and $B \neq []$ imply there is a configuration d_1 and a store type Σ_1 such that $(a', \sigma_0') \Downarrow d_1$, $\vdash (\iota, \sigma_1) \leftrightarrow d_1 : (B, \Sigma_1)$

and $\Sigma \leq \Sigma_1$. Hence $d_1 = (\iota, \sigma'_1)$ with $([], \Sigma_1) \vdash \iota \leftrightarrow \iota : B$, which implies that $\iota \in dom(\Sigma_1)$, $\Sigma_1(\iota) = B$ and $\vdash \sigma_1 \leftrightarrow \sigma'_1 : \Sigma_1$. By the latter, $([], \Sigma_1) \vdash \sigma_1(\iota) \leftrightarrow \sigma'_1(\iota) : B$. The latter implies that $layout([\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]) = B$, and therefore that $B = [\ell_i\ ^{i \in 1..n}]$, $m = n$ and each $\ell_i = \ell'_i$. It also implies there is o' with $\sigma'_1(\iota) = o'$ and $o' = [\ell_i = \varsigma(x_i)b'_i\ ^{i \in 1..n}]$. By (Layout Object), there is A_j with $([x_j \mapsto B], \Sigma_1) \vdash b_j \leftrightarrow b'_j : A_j$. By Lemma 5.2, $\iota \in dom(\Sigma_1)$ implies $([], \Sigma_1) \vdash b_j\{\!\!\{\iota/x_j\}\!\!\} \leftrightarrow b'_j\{\!\!\{\iota/x_j\}\!\!\} : A_j$. By (Layout Config), $\vdash (b_j\{\!\!\{\iota/x_j\}\!\!\}, \sigma_1) \leftrightarrow (b'_j\{\!\!\{\iota/x_j\}\!\!\}, \sigma'_1) : (A_j, \Sigma_1)$. By induction hypothesis, the latter and $(b_j\{\!\!\{\iota/x_j\}\!\!\}, \sigma_1) \Downarrow c'$ imply there are d', A' and Σ' such that $(b'_j\{\!\!\{\iota/x_j\}\!\!\}, \sigma'_1) \Downarrow d'$, $\vdash c' \leftrightarrow d' : (A', \Sigma')$ and $\Sigma_1 \leq \Sigma'$. By (Subst Offset Select), we have $c' = (a'.j, \sigma'_0) \Downarrow d'$. We have $\Sigma \leq \Sigma'$ from $\Sigma \leq \Sigma_1$ and $\Sigma_1 \leq \Sigma'$, since $\leq$ is clearly transitive. Finally, $A \neq []$ implies $A = A'$ holds vacuously, since $A = []$.

The cases for (Layout Select 1) and (Layout Select 2) are very similar.

We omit the remaining cases, which are no harder than the one shown. The case for (Subst Update) is similar to the one shown. The cases for (Subst Offset Select) and (Subst Offset Update) are slightly simpler than (Subst Select) and (Subst Update) respectively. The remaining cases are routine. $\square$

Lemma 5.4 *Suppose* $([x_i \mapsto []\ ^{i \in 1..n}], []) \vdash a \leftrightarrow a' : A$. *Consider any reduction context* $\mathcal{R}$ *with* $locs(\mathcal{R}) = \varnothing$ *such that* $fv(\mathcal{R}) - \{\bullet, x_1, \ldots, x_n\} = \{x_{n+1}, \ldots, x_{n+m}\}$. *Then* $([x_i \mapsto []\ ^{i \in 1..n+m}], []) \vdash \mathcal{R}[a] \leftrightarrow \mathcal{R}[a'] : B$ *for some* B.

Proof By induction on the size of the reduction context $\mathcal{R}$, with appeal to rules (Layout Select 1), (Layout Select 2), (Layout Update 1), (Layout Update 2), (Layout Clone) and (Layout Let). Moreover, we need the facts that the $\leftrightarrow$ relation is reflexive (if $locs(a) \subseteq dom(\Sigma)$ and $fv(a) \subseteq dom(E)$ then $(E, \Sigma) \vdash a \leftrightarrow a : A$ holds for some A) and satisfies environment weakening (if $dom(E) \subseteq dom(E')$ and $E(x) = E'(x)$ for each $x \in dom(E)$, then $(E, \Sigma) \vdash a \leftrightarrow a' : A$ implies $(E', \Sigma) \vdash a \leftrightarrow a' : A$). $\square$

Lemma 5.5 *Given* $([x_i \mapsto []\ ^{i \in 1..n}], []) \vdash a \leftrightarrow a' : B$, *a store type* Σ *and a substitution* $\rho : \{x_1, \ldots, x_n\} \to dom(\Sigma)$, *there is* B' *such that* $([], \Sigma) \vdash a\rho \leftrightarrow a'\rho : B'$. *Moreover,* $B \neq []$ *implies* $B = B'$.

Proof By induction on the derivation of $([x_i \mapsto []\ ^{i \in 1..n}], []) \vdash a \leftrightarrow a' : B$. $\square$

Theorem 5.6 *Suppose a is a static term with free variables $x_1,\dots,x_n$. If routine resolve$([x_1 \mapsto []\ ^{i \in 1..n}], a)$ returns (a', A), then $a \approx a'$.*

Proof By Lemma 4.5, to show $a \approx a'$, it suffices to prove $(\mathcal{R}[a]\rho, \sigma) \updownarrow (\mathcal{R}[a']\rho, \sigma)$, for all static reduction contexts $\mathcal{R}$, well formed stores σ, and substitutions $\rho : \mathit{fv}(\mathcal{R}[a]) \cup \mathit{fv}(\mathcal{R}[a']) \to \mathit{dom}(\sigma)$. Consider any static reduction context $\mathcal{R}$, any well formed store σ and any substitution $\rho : \mathit{fv}(\mathcal{R}[a]) \cup \mathit{fv}(\mathcal{R}[a']) \to \mathit{dom}(\sigma)$. Let $E = [x_1 \mapsto []\ ^{i \in 1..n}]$ and $E' = [x_1 \mapsto []\ ^{i \in 1..n+m}]$ where $\{x_{n+1}, \dots, x_{n+m}\} = \mathit{fv}(\mathcal{R}) - \{\bullet, x_1, \dots, x_n\}$. By Lemma 5.1, we may derive $(E, []) \vdash a \leftrightarrow a' : A$. By Lemma 5.4, $(E, []) \vdash a \leftrightarrow a' : A$ implies $(E', []) \vdash \mathcal{R}[a] \leftrightarrow \mathcal{R}[a'] : B$ for some B. If $\sigma = [\iota_i = o_i\ ^{i \in 1..n}]$, let $\Sigma = [\iota_i = \mathit{layout}(o_i)\ ^{i \in 1..n}]$. By Lemma 5.5, $(E', []) \vdash \mathcal{R}[a] \leftrightarrow \mathcal{R}[a'] : B$ and $\rho : \{x_1, \dots, x_{n+m}\} \to \mathit{dom}(\Sigma)$ imply $([], \Sigma) \vdash \mathcal{R}[a]\rho \leftrightarrow \mathcal{R}[a']\rho : B'$ for some B'. By (Layout Store), $\Sigma \vdash \sigma \leftrightarrow \sigma$. Hence by (Layout Config), we have $\vdash (\mathcal{R}[a]\rho, \sigma) \leftrightarrow (\mathcal{R}[a']\rho, \sigma)$. Suppose that $(\mathcal{R}[a]\rho, \sigma)\downarrow$. By Theorem 2.7 there is c with $(\mathcal{R}[a]\rho, \sigma) \Downarrow c$. By Proposition 5.3(1), $\vdash (\mathcal{R}[a]\rho, \sigma) \leftrightarrow (\mathcal{R}[a']\rho, \sigma)$ implies there is c' such that $(\mathcal{R}[a']\rho, \sigma) \Downarrow c'$, and therefore $(\mathcal{R}[a']\rho, \sigma)\downarrow$, again by Theorem 2.7. Similarly, by Proposition 5.3(2) and $\vdash (\mathcal{R}[a]\rho, \sigma) \leftrightarrow (\mathcal{R}[a']\rho, \sigma)$, $(\mathcal{R}[a']\rho, \sigma)\downarrow$ implies $(\mathcal{R}[a]\rho, \sigma)\downarrow$. Therefore $(\mathcal{R}[a]\rho, \sigma) \updownarrow (\mathcal{R}[a']\rho, \sigma)$, as required to establish that $a \approx a'$. □

Our prototype implementation of the imperative object calculus optimises any closed static term a by running the routine *resolve*$([], a)$ to obtain an optimised term a' paired with a layout type A. By the theorem, this optimisation is correct in the sense that a' is operationally equivalent to a. In fact the theorem applies to applications of the *resolve* routine to open terms. Inasmuch as we may regard a module as a term with free variables, the theorem would justify use of *resolve* during separate compilation of modules.

On a limited set of test programs, the algorithm converts a majority of selects and updates into the optimised form. However, the speedup ranges from modest (10%) to negligible; the interpretive overhead in our bytecode-based system tends to swamp the effect of optimisations such as this. It is likely to be more effective in a native code implementation.

5.4 Discussion and Related Work

In general, there are many algorithms for optimising access to objects; see Chambers (1992), for instance, for examples and a literature survey. The idea of statically resolving labels to integer offsets is found also in the work of Ohori (1992), who presents a λ-calculus with records and a polymorphic type system such that a compiler may compute integer offsets for all uses of record labels. Our system is rather different, in that it exploits object-oriented references to self.

In contrast to Ohori's type system, we have not integrated our layout types with a conventional type system that guarantees the absence of unchecked runtime errors. Our system of layout types could probably be integrated with one or other of Abadi and Cardelli's type systems for the imperative object calculus, to obtain a unified type system that avoided unchecked runtime errors and moreover could determine statically the layout of certain objects. Instead, our implementation checks programs using one of Abadi and Cardelli's type systems in one pass, and in a separate pass uses the algorithm from this section to optimise updates and selects. This separation avoids the complications of a unified type system.

6 Conclusions

In this report, we have collated and extended a range of operational techniques which we have used to verify aspects of the implementation of a small object-oriented programming language, Abadi and Cardelli's imperative object calculus.

Our first result is a correctness proof for an abstract machine and its compiler, Theorem 3.18. Such results are rather more difficult than proofs of interpretive abstract machines. Our contribution is a direct proof method which avoids the need for any metalanguage—such as a calculus of explicit substitutions. Our second result is that Mason and Talcott's CIU equivalence coincides with Morris-style contextual equivalence, Theorem 4.13. The benefit of CIU equivalence is that it allows the verification of compiler optimisations. We illustrate this by proving Theorem 5.6, which asserts that an optimisation algorithm from our implementation preserves contextual equivalence.

This is the first study of correctness of compilation to an object-oriented abstract machine. It is also the first study of program equivalence for the imperative object calculus, a topic left unexplored by Abadi and Cardelli's book. To the best of our knowledge, the only other work on the imperative object calculus is a program logic due to Abadi and Leino (1997) and a brief presentation, without discussion of equivalence, of a labelled transition system for untyped imperative objects in the thesis of Andersen and Pedersen (1996).

Abadi and Cardelli's object calculi are based on fixed size objects. A λ-calculus of objects in which objects may be dynamically extended with new methods was proposed by Mitchell, Honsell, and Fisher (1993); we conjecture that our techniques could easily be adapted to an imperative form of their calculus.

Our motivation is to promote language description based on operational semantics. By means of a big language, Standard ML, Milner, Tofte, and Harper (1990) and more recently Harper and Stone (1996) have contributed substantially to this project by demonstrating the possibility of an operational description of a practical programming language. By means of a little language, the imperative object calculus, our contribution is to collate a wide range of styles of formal description, and to demonstrate how properties may be proved of these descriptions.

Acknowledgements

The static resolution algorithm of Section 5 arose in discussions with Roy Crole. Martín Abadi and Carolyn Talcott commented on a draft. Gordon holds a Royal Society University Research Fellowship. Hankin holds an EPSRC Research Studentship. Lassen is supported by a grant from the Danish National Science Research Council.

References

Abadi, M. and L. Cardelli (1995a). An imperative object calculus. In *TAPSOFT'95: Theory and Practice of Software Development*, Volume 915 of *Lecture Notes in Computer Science*, pp. 471–485. Springer-Verlag.

Abadi, M. and L. Cardelli (1995b). An imperative object calculus: Basic typing and soundness. In *SIPL '95 - Proceedings of the Second ACM SIGPLAN Workshop on State in Programming Languages*. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign.

Abadi, M. and L. Cardelli (1996). *A Theory of Objects*. Springer-Verlag.

Abadi, M. and R. Leino (1997). A logic for object-oriented programs. In *TAPSOFT'97*. To appear.

Abelson, H. and G. J. Sussman (1985). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass.

Agha, G. A., I. A. Mason, S. F. Smith, and C. L. Talcott (1997, January). A foundation for actor computation. *Journal of Functional Programming 7*(1).

Andersen, D. S. and L. H. Pedersen (1996). An operational approach to the ς-calculus. Master's thesis, Department of Mathematics and Computer Science, Aalborg. Available as Report R–96–2034.

Andersen, D. S., L. H. Pedersen, H. Hüttel, and J. Kleist (1997, January). Object types and modal formulae. In *FOOL 4, Paris*.

Cardelli, L. (1995, January). A language with distributed scope. *Computing Systems 8*(1), 27–59.

Chambers, C. (1992, March). *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph. D. thesis, Computer Science Department, Stanford University.

Felleisen, M. (1995, February). Programming languages and lambda calculi. URL:http://www.cs.rice.edu/ matthias/411web/mono.ps. Unpublished course notes, Rice University.

Felleisen, M. and D. Friedman (1986). Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pp. 193–217. North-Holland.

Gordon, A. D. (1994). *Functional Programming and Input/Output*. Cambridge University Press.

Gordon, A. D. (1997). Operational equivalences for untyped and polymorphic object calculi. In *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute. Cambridge University Press. To appear.

Gordon, A. D. and G. D. Rees (1996). Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of the Twenty Third ACM Symposium on Principles of Programming Languages*, pp. 386–395. ACM. Accepted for publication in *Information and Computation*.

Guttman, J. D., V. Swarup, and J. Ramsdell (1995). The VLISP verified scheme system. *Lisp and Symbolic Computation 8*(1/2), 33–110.

Hannan, M. (1992). From operational semantics to abstract machines. *MSCS 4*(2), 415–489.

Hardin, T., L. Maranget, and B. Pagano (1996, May). Functional backends and compilers within the lambda-sigma calculus. In *ICFP'96*.

Harper, R. and C. Stone (1996). A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU–CS–96–136R, Computer Science Department, Carnegie-Mellon University.

Honsell, F., I. A. Mason, S. Smith, and C. Talcott (1993). A variable typed logic of effects. *Information and Computation 119*(1), 55–90.

Howe, D. J. (1996, February). Proving congruence of bisimulation in functional programming languages. *Information and Computation 124*(2), 103–112.

Hüttel, H. and J. Kleist (1996). Objects as mobile processes. In *Proceedings MFPS'96*.

Jones, C. B. (1995). Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*. To appear.

Kahn, G. (1988). Natural semantics. In K. Fuchi and M. Nivat (Eds.), *Programming of Future Generation Computers*, pp. 237–258. Elsevier Science Publishers B.V. (North-Holland). Appeared as Rapport de Recherche no. 601, INRIA.

Landin, P. (1964). The mechanical evaluation of expressions. *Computer Journal 6*, 308–320.

Leroy, X. (1990). The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA.

Martin-Löf, P. (1983, August). Notes on the domain interpretation of type theory. Proceedings of the Workshop on Semantics of Programming Languages, Chalmers, 1983. See also Nordström, Petersson, and Smith (1990).

Mason, I. A. and C. L. Talcott (1991). Equivalence in functional languages with effects. *Journal of Functional Programming 1*(3), 287–327.

Mason, I. A. and C. L. Talcott (1992). References, local variables and operational reasoning. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*.

Mason, I. A. and C. L. Talcott (1994). Reasoning about object systems in VTLoE. Submitted to *International Journal of Foundations of Computer Science*.

McCarthy, J. (1963). A basis for a mathematical theory of computation. In P. Braffort and D. Hirshberg (Eds.), *Computer Programming and Formal Systems*. North-Holland. URL:http://www-formal.stanford.edu/jmc/basis.html.

Meyer, A. and K. Sieber (1988). Towards fully abstract semantics for local variables. In *POPL'88*.

Meyer, A. R. and S. S. Cosmadakis (1988, July). Semantical paradigms: Notes for an invited lecture. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pp. 236–253.

Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML*. MIT Press, Cambridge, Mass.

Mitchell, J. C., F. Honsell, and K. Fisher (1993). A lambda calculus of objects and method specialization. In *Proceedings of the Eighth IEEE Symposium on Logic in Computer Science, Montreal*, pp. 26–38.

Moggi, E. (1989). Notions of computations and monads. *Information and Computation 93*, 55–92. Earlier version in LICS'89.

Morris, J. H. (1968, December). *Lambda-Calculus Models of Programming Languages*. Ph. D. thesis, MIT.

Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf's Type Theory*, Volume 7 of *The International Series of Monographs in Computer Science*. Clarendon Press, Oxford.

Ohori, A. (1992). A compilation method for ML-style polymorphic record calculi. In *Proceedings 19th ACM Symposium on Principles of Programming Languages*, pp. 154–165. ACM.

Pitts, A. M. (1997). Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn and R. D. Tennent (Eds.), *Algol-Like Languages*, Volume 2, Chapter 17, pp. 173–193. Birkhauser. Reprinted from *Proceedings Eleventh Annual IEEE Symposium on Logic in Computer Science*, Brunswick, NJ, July 1996, pp 152–163.

Plotkin, G. (1975). Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science 1*, 125–159.

Plotkin, G. D. (1977). LCF considered as a programming language. *Theoretical Computer Science 5*, 223–255.

Plotkin, G. D. (1981, September). A structural approach to operational semantics. Technical Report FN–19, DAIMI, Aarhus University.

Rittri, M. (1990). *Proving compiler correctness by bisimulation*. Ph. D. thesis, Chalmers.

Sangiorgi, D. (1996). An interpretation of typed objects into typed π-calculus. In *FOOL 3, New Brunswick*.

Sangiorgi, D. (1997). Typed π-calculus at work: a proof of Jones' parallelisation transformation on concurrent objects. In *FOOL 4, Paris*.

Sestoft, P. (1994, September). Deriving a Lazy Abstract Machine. ID-TR 1994-146, Department of Computer Science, Technical University of Denmark.

Talcott, C. (1997). Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute. Cambridge University Press. To appear.

Walker, D. (1995, February). Objects in the pi-calculus. *Information and Computation 116*(2), 253–271.

Wand, M. (1995, June). Compiler correctness for parallel languages. In *Functional Programming and Computer Architecture*, pp. 120–134. ACM.