J. Functional Programming **9** (4): 373–426, July 1999. Printed in the United Kingdom
© 1999 Cambridge University Press

Compilation and equivalence of imperative objects

ANDREW D. GORDON*, PAUL D. HANKIN†
University of Cambridge Computer Laboratory, University of Cambridge, Cambridge, UK

SØREN B. LASSEN‡
BRICS§, Department of Computer Science, University of Aarhus, Aarhus, Denmark

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. Our first two results are theorems asserting the equivalence of our substitution-based semantics with a closure-based semantics like that given by Abadi and Cardelli. Our third result is a direct proof of the correctness of compilation to a stack-based abstract machine via a small-step decompilation algorithm. Our fourth 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.

Capsule Review

This paper presents a rigorous semantic study of the imperative object calculus of Abadi and Cardelli presenting: several forms of operational semantics along with correspondence theorems; an abstract machine, compiler and correctness result; and a theory of operational equivalence and its application to reasoning about program equivalence and correctness of program optimizations.

The paper is well-written and a pleasure to read. It brings together a number of existing techniques in operational semantics for imperative functional languages, making a coherent whole and providing a guideline for semantics based development of object-oriented languages. Adding additional data types and program syntax (conditional, loops, etc.) should be a matter of filling in details following the guidelines given. This is likely to be a frequently cited paper. The paper sets the stage for a number of important future developments including extension of the study to concurrent objects and investigation of additional forms of program analysis and optimization.

* Current affiliation: Microsoft Research. E-mail: `adg@microsoft.com`
† E-mail: `pdh13@cam.ac.uk`
‡ Current affiliation: University of Cambridge Computer Laboratory.
E-mail: `Soeren.Lassen@cl.cam.ac.uk`
§ Basic Research in Computer Science, Centre of the Danish National Research Foundation.

1 Introduction

This paper collates and extends a variety of operational techniques for describing and reasoning about programming languages and their implementation. We focus on implementation of imperative object-oriented programs, expressed in an imperative object calculus. We examine different forms of structural operational semantics for the calculus, specify an implementation in terms of an object-oriented abstract machine, and develop a theory of operational equivalence between programs which we use to specify and verify a simple compiler optimisation. Many of our semantic techniques originate in earlier studies of the λ-calculus. This paper is their first application to an object calculus and shows they may easily be re-used in an object-oriented setting.

The language we describe is essentially the untyped imperative object calculus of Abadi and Cardelli (1995*a*; 1995*b*; 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 broader project to investigate object-oriented languages. Other work considers a concurrent variant of the imperative object calculus (Gordon and Hankin, 1998). 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[1] of Leroy's CAML Light (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. Since the results of the paper are independent of this type system, we will say no more about it.

The rest of the paper is organised as follows:

- In section 2 we present our source language, the imperative object calculus, together with three forms of operational semantics (Plotkin, 1981; Martin-Löf, 1983; Felleisen and Friedman, 1986; Kahn, 1987). Theorems 1 and 2 assert the consistency of these semantics.
- 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 a compiler correctness result, Theorem 3, by adapting an idea of Rittri (1990) to cope with state and objects.
- 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

[1] 'ZAM' is an acronym for 'Zinc Abstract Machine', where 'Zinc' is an acronym for 'Zinc is not Caml'.

become the standard for studies of λ-calculi, to the imperative object calculus. Our fourth result, Theorem 4, characterises contextual equivalence using the CIU equivalence of Mason and Talcott (1991).

- In section 5, we exercise operational equivalence by specifying a simple optimisation that resolves at compile-time certain method labels to integer offsets. Theorem 5 states the correctness of the optimisation.

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.

Mostly, we state propositions without proofs or with brief proof outlines only. Some technical lemmas needed for the proofs are omitted. These and the full proofs may be found in a technical report (Gordon *et al.*, 1998).

2 An imperative object calculus

In this section, we present the syntax of an imperative object calculus, together with three forms of operational semantics. Theorems 1 and 2 state the equivalence of these semantics.

2.1 Syntax of the calculus

We begin with the syntax of an untyped imperative object calculus, the **imp**ς 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*, ℓ range over an infinite collection of *method labels*, and 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 an existing method in a stored object to be updated. 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 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 do not identify objects up to re-ordering of methods. This is because the order of methods in an object is significant for an application of our techniques presented in section 5. Moreover, we include locations in the syntax of terms. This is so we may express the dynamic behaviour of the calculus using a substitution-based operational semantics. In Abadi and Cardelli's closure-based semantics, locations appear only in closures and not in terms. 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. Terms containing locations arise during reduction.

As a first example of programming in the imperative object calculus, here is how to express pairs of terms as objects with *fst* and *snd* methods for accessing the two components and a *swap* method for interchanging the first and second components:

$$\begin{aligned} pair(a,b) \stackrel{\text{def}}{=} \; & [fst = \varsigma(s)a, \\ & snd = \varsigma(s)b, \\ & swap = \varsigma(s)let\ \ x = s.fst\ \ in \\ & \qquad\qquad let\ \ y = s.snd\ \ in \\ & \qquad\qquad\quad (s.fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')x] \\ & \text{for } s \notin fv(a) \cup fv(b) \end{aligned}$$

The next example makes use of the imperative nature of the calculus to express updateable references as 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 third example, here is an encoding of the call-by-value λ-calculus:

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

where $y \neq z$, and s and y do not occur free in b. It is like an encoding from Abadi and Cardelli's book but with right-to-left evaluation of function application. Given updateable methods, we can easily extend this encoding to express an ML-style call-by-value λ-calculus with updateable references.

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 an efficient 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
$\dots$	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 $[\phi_1, \dots, \phi_n]$, 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 $dom(f) = \{x_i\ ^{i \in 1..n}\}$. For the finite map $f = f' @ [x \mapsto \phi] @ f''$, let $f(x) = \phi$. When f is a finite map, 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 refer to such relations as reduction or transition 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, $\mathscr{R}$, is a term given by the following grammar, with one free occurrence of a distinguished variable, $\bullet$, which represents 'the point of execution' in $\mathscr{R}$.

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

Since there is exactly one free occurrence of $\bullet$ in any reduction context, if $\mathscr{R}.\ell \Leftarrow \varsigma(x)b$ is a reduction context, $\bullet \notin fv(b) - \{x\}$. For the same reason, if $let\ \ x = \mathscr{R}\ in\ b$, $a(\mathscr{R})$, and $\mathscr{R}(v)$ are reduction contexts, $\bullet \notin fv(b) - \{x\}$, $\bullet \notin fv(a)$ and $\bullet \notin fv(v)$, respectively. We write $\mathscr{R}[a]$ for the outcome of substituting term a (not necessarily a value) for the single occurrence of the hole $\bullet$ in a reduction context $\mathscr{R}$. 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, $c \rightarrow d$, be the least relation satisfying the following axiom schemes:

(Red Object) $(\mathscr{R}[o], \sigma) \rightarrow (\mathscr{R}[\iota], \sigma')$ if $\sigma' = (\iota \mapsto o) :: \sigma$ and $\iota \notin dom(\sigma)$.
(Red Select) $(\mathscr{R}[\iota.\ell_j], \sigma) \rightarrow (\mathscr{R}[b_j\{\!\{\iota/x_j\}\!\}], \sigma)$
if $\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]$ and $j \in 1..n$.
(Red Update) $(\mathscr{R}[\iota.\ell_j \Leftarrow \varsigma(x)b], \sigma) \rightarrow (\mathscr{R}[\iota], \sigma')$
if $\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]$, $j \in 1..n$, and
$\sigma' = \sigma + (\iota \mapsto [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i\ ^{i \in j+1..n}])$.
(Red Clone) $(\mathscr{R}[clone(\iota)], \sigma) \rightarrow (\mathscr{R}[\iota'], \sigma')$
if $\sigma(\iota) = o$, $\sigma' = (\iota' \mapsto o) :: \sigma$ and $\iota' \notin dom(\sigma)$.
(Red Let) $(\mathscr{R}[let\ \ x = v\ in\ b], \sigma) \rightarrow (\mathscr{R}[b\{\!\{v/x\}\!\}], \sigma)$.
(Red Appl) $(\mathscr{R}[(\lambda(x)b)(v)], \sigma) \rightarrow (\mathscr{R}[b\{\!\{v/x\}\!\}], \sigma)$.

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 1
Suppose $\vdash (a, \sigma)\ ok$ and $(a, \sigma) \rightarrow (a', \sigma')$. Then $\vdash (a', \sigma')\ ok$ and $dom(\sigma) \subseteq dom(\sigma')$.

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 $c \rightarrow^* d$. 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:

$$(pair(\iota_1, \iota_2).swap, \sigma)$$

where σ is a well formed store of the form $[\iota_1 \mapsto o_1, \iota_2 \mapsto o_2]$ and *pair* is as defined in section 2.1. This is not a terminal configuration, but it converges because of the following reduction sequence (in which we assume $\iota \notin dom(\sigma)$).

$$\begin{array}{ll}
\multicolumn{2}{l}{(pair(\iota_1, \iota_2).swap, \sigma)} \\
\to & (\iota.swap, (\iota \mapsto pair(\iota_1, \iota_2)) :: \sigma) \\
\to & (let\ x = \iota.fst\ in\ let\ y = \iota.snd\ in\ (\iota.fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')x, \\
& \quad (\iota \mapsto pair(\iota_1, \iota_2)) :: \sigma) \\
\to & (let\ x = \iota_1\ in\ let\ y = \iota.snd\ in\ (\iota.fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')x, \\
& \quad (\iota \mapsto pair(\iota_1, \iota_2)) :: \sigma) \\
\to & (let\ y = \iota.snd\ in\ (\iota.fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')\iota_1, \\
& \quad (\iota \mapsto pair(\iota_1, \iota_2)) :: \sigma) \\
\to & (let\ y = \iota_2\ in\ (\iota.fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')\iota_1, \\
& \quad (\iota \mapsto pair(\iota_1, \iota_2)) :: \sigma) \\
\to & ((\iota.fst \Leftarrow \varsigma(s')\iota_2).snd \Leftarrow \varsigma(s')\iota_1, (\iota \mapsto pair(\iota_1, \iota_2)) :: \sigma) \\
\to & (\iota.snd \Leftarrow \varsigma(s')\iota_1, (\iota \mapsto pair(\iota_2, \iota_2)) :: \sigma) \\
\to & (\iota, (\iota \mapsto pair(\iota_2, \iota_1)) :: \sigma)
\end{array}$$

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 $\vdash_w (a, \sigma)\ ok$ 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) $\dfrac{\vdash_w c\ ok}{c \equiv_w c}$ **(Struct Trans)** $\dfrac{c \equiv_w c' \quad c' \equiv_w c''}{c \equiv_w c''}$ **(Struct Symm)** $\dfrac{c \equiv_w c'}{c' \equiv_w c}$

(Struct Rename)

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

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 $\vdash_w c\ ok$ and $\vdash_w d\ ok$. 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.

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

Proposition 2
Suppose $\vdash_w c\ ok$. Then $c \rightarrow c'$ and $c \rightarrow c''$ imply $c' \equiv_w c''$.

Furthermore, reduction and structural equivalence possess the following property.

Lemma 3
Suppose $c \equiv_w c'$. Then $c \rightarrow d$ implies there exists d' such that $c' \rightarrow d'$ and $d \equiv_w d'$.

Proposition 2 and Lemma 3 imply that whenever (a,σ) is well formed and $(a,\sigma) \rightarrow^* d$, 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.

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, $\rightarrow$, of the previous section. Unlike the $\rightarrow$ relation, the $\Downarrow$ relation is defined inductively. We exploit its induction principle in the proof of Proposition 38, 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)

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

(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 $j \in 1..n$)

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

(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)

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

We define $c \searrow d$ to mean that $c \rightarrow^* d$ and d is terminal. The big-step and small-step substitution semantics are consistent with one another in the following sense:

Theorem 1

(1) Whenever $c \Downarrow d$, $c \searrow d$.

(2) Whenever $c \searrow d$, $c \Downarrow d$.

Proof

Part (1) is by induction on the derivation of $c \Downarrow d$. For part (2), one can prove by induction on n that $c \Downarrow d$ whenever $c \rightarrow^n d$ and d is terminal. $\square$

It follows, by the results about the small-step relation in Proposition 2 and Lemma 3, that the big-step relation, $\Downarrow$, is also deterministic up to structural equivalence:

Proposition 4

Whenever $\vdash_w c\ ok$, $c \Downarrow c'$ and $c \Downarrow c''$ imply $c' \equiv_w c''$.

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 *et al.*, 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
$\quad \iota$	location
$\quad (S, \lambda(x)b)$	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
$\quad ((S, a), \Sigma)$	initial configuration
$\quad (V, \Sigma)$	terminal configuration

A stack (of bindings) $S = [x_i \mapsto V_i \; ^{i \in 1..n}]$ 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. A configuration of the form (V, Σ) where $V = (S, \lambda(x)b)$ is both initial and terminal.

Our syntax admits stores and configurations that include dangling pointers and unbound variables. We could make an explicit definition of those well formed stores and configurations that do not include such errors. Instead, it is more convenient, later on in this section, to make an implicit definition of well formed stores and configurations in terms of an unloading relation.

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 closure-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)

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

(Closure Update)

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

(Closure Object)

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

(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)

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

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 5
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 $\rightsquigarrow$ for each of five unloading relations.

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

(Value ι) **(Value Fun)**

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

(Stack []) **(Stack Object)**

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

(Object Unload) (where ℓ_i distinct)

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

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

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

(Config Initial) **(Config Terminal)**

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

The unloading relations possess the following properties.

Proposition 6

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

(2) Whenever $S \rightsquigarrow s$ 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 $O \rightsquigarrow o$, object o is closed.

(4) Whenever $\Sigma \rightsquigarrow \sigma$, 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. □

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 $C \rightsquigarrow c$. The only occurrences of locations in a well formed configuration are in the domain of the store and in the range of any stacks occurring in the configuration.

The unloading relations are in fact functional:

Proposition 7
Whenever $\phi \rightsquigarrow \psi'$ and $\phi \rightsquigarrow \psi''$, then $\psi' = \psi''$.

To prove Theorem 2, which asserts the consistency of the two big-step operational semantics, we need the following two lemmas.

Lemma 8
If $C \rightsquigarrow c$ and $C \Downarrow C'$ there is c' such that $C' \rightsquigarrow c'$ and $c \Downarrow c'$.

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

Lemma 9
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 $c \Downarrow c'$. □

Theorem 2
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 8 there is c'' with $C' \rightsquigarrow c''$ and $c \Downarrow c''$. By Proposition 7, $c' = c''$. On the other hand, suppose $c \Downarrow c'$. By Lemma 9, there is a terminal configuration C'' such that $C'' \rightsquigarrow c'$ and $C \Downarrow C''$. By Proposition 7, $C' = C''$. □

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 1, 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, 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). There is a proof by Felleisen and Friedman (1989) of the equivalence of substitution-based and closure-based semantics for an imperative λ-calculus, but they work with small-step rather than big-step semantics.

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 correctness result, Theorem 3. 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 was designed for efficient evaluation of curried functions. The machine configuration consists of a state paired with a store. A store is a finite map from locations to stored objects. A state is a quadruple, (ops, AS, E, RS), consisting of a list of instructions (or operations), ops, an argument stack, AS, an environment, E, and a return stack, RS. The instruction list is obtained from compiling some source term. Each item on the argument stack is either a value, V, or a mark, $\diamond$. A value is either the location, ι, of an object in the store, or a closure, (ops, E), which is an operation list ops paired with an environment E. A mark is a special tag introduced by Leroy for efficient evaluation of functions. An environment is a list of values that represents the runtime values assumed by variables free in the original source term. The return stack is a list of frames representing the currently active method invocations and function calls. A frame is simply a closure.

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 (curried) arguments off the stack, and discovers when it has consumed all its arguments when it finds the mark. If the function returns (on executing a `return` instruction) and 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`	apply function
`grab`	get curried argument
`pushmark`	push mark onto stack
`return`	return from function

$ops ::= [] \mid op :: ops$

We describe the workings of our machine informally as follows:

- The instruction `access` i fetches the ith value in the current environment, and pushes it onto the argument stack. It is used for looking up the value of a variable.
- The instruction `object`$[(\ell_i, ops_i)^{\,i\in 1..n}]$ creates a new object in the store, and pushes the location of the newly created object onto the argument stack. The ℓ_i are method labels and the ops_i are the corresponding compiled methods.
- The instruction `select`ℓ pops the location of an object off the argument stack, and loads from the object the method closure (ops, E) labelled ℓ. The current operation list and environment are saved by pushing them as a pair onto the return stack, and then are replaced by the new operation list ops and the new environment E.
- The instruction `update`(ℓ, ops) pops the location of an object off the argument stack, and updates the method closure labelled ℓ in that object with the closure (ops, E), where E is the current environment.
- The instruction `let`ops pops a value off the argument stack, and adds it to the environment. The instructions ops are then executed in the new environment. A frame built from the remainder of the operation list and the current environment is pushed onto the return stack, to be executed once the instructions ops have been completed.
- The instruction `cur` ops pushes a function closure onto the argument stack. The closure is built by pairing the compiled function body, ops, with the current environment.
- The instruction `apply` pops a function closure and value off the argument stack. The current operation list and environment are pushed as a frame onto the return stack, and the closure is executed with the value (the argument to the function) added to its environment.
- The instruction `pushmark` pushes a mark, $\diamond$, onto the argument stack. This instruction is used to delimit a series of curried arguments to a function.
- The instruction `grab` examines the top of the argument stack. If the top of the argument stack is a mark, $\diamond$, the `grab` instruction builds the current state into a closure and returns to the function caller by popping a frame off the return stack. Otherwise, if the top of the argument stack is a value, the value is added to the environment and the execution of the function proceeds. The `grab` instruction starts the compiled form of a nested function. For example,

in the term $\lambda(x)\lambda(y)a$, the compilation of the $\lambda(y)a$ term will start with a `grab` instruction.

- The instruction `return` can be considered a dual to `grab`. When `return` is executed (at the end of a function call), the value the function is returning is on the top of the argument stack. If the return value is a function, and this function is being applied directly to an argument, the value will be second on the argument stack. In this case, `return` will perform the function application without returning to the original function caller. On the other hand, if the return value is not being applied to an argument, a mark, $\diamond$, will be second on the argument stack. In this case, the mark is removed and the function caller is popped back off the return stack.

We now give a formal definition of the abstract machine. 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
$U^{\diamond}, V^{\diamond} ::= U \mid \diamond$	value or mark
$E ::= [U_i{}^{i \in 1..n}]$	environment
$AS ::= [U_i^{\diamond}{}^{i \in 1..n}]$	argument stack
$RS ::= [F_i{}^{i \in 1..n}]$	return stack
$F ::= (ops, E)$	closure or frame
$O ::= [(\ell_i, F_i)^{i \in 1..n}]$	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, $\diamond$. 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 18 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) $(([\texttt{return}], E, fun(ops, E') :: U :: AS, RS), \Sigma) \xrightarrow{\beta}$
$((ops, U :: E', AS, RS), \Sigma)$.
(τ Grab) $((\texttt{grab} :: ops, E, \diamond :: AS, (ops', E') :: RS), \Sigma) \xrightarrow{\tau}$
$((ops', E', fun(ops, E) :: AS, RS), \Sigma)$.
(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) $((\texttt{cur}\ ops :: ops', E, AS, RS), \Sigma) \xrightarrow{\tau}$
$((ops', E, fun(ops, E) :: AS, RS), \Sigma)$.

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

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

We now describe compilation of the object calculus to the instruction set of our abstract machine. We use the notation $\texttt{grab}^n$ for the list $[\texttt{grab}, \texttt{grab}, \ldots, \texttt{grab}]$ consisting of n $\texttt{grab}$ instructions, and the notation $\lambda(x_1 x_2 \ldots x_n)a$ for the term $\lambda(x_1)\lambda(x_2)\ldots\lambda(x_n)a$ when $n > 0$ and a when $n = 0$. 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) $xs \vdash \lambda(x_{n+1} x_n \ldots x_1)a \Rightarrow [\texttt{cur}(\texttt{grab}^n @ ops @ [\texttt{return}])]$
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 $[x_i{}^{\,i \in 1..n+1}] @ xs \vdash a \Rightarrow ops$.

3.2 Examples of compilation and execution

We illustrate compilation and execution via three examples.

Example 1: Method invocation

As a first example, let the term $a = pair([],[]).fst$, ($pair$ was defined in section 2.1). We have $[] \vdash a \Rightarrow ops$, where the operation list ops is given by:

$$\begin{aligned}
ops &= [\texttt{object}[(fst, ops_1), (snd, ops_2), (swap, ops_3)], \texttt{select}\, fst] \\
ops_1 &= [\texttt{object}[]] \\
ops_2 &= [\texttt{object}[]] \\
ops_3 &= [\texttt{access}\ 1, \texttt{select}\, fst, \texttt{let}\ ops_4] \\
ops_4 &= [\texttt{access}\ 2, \texttt{select}\ snd, \texttt{let}\ ops_5] \\
ops_5 &= [\texttt{access}\ 3, \texttt{update}(fst, [\texttt{access}\ 2]), \texttt{update}(snd, [\texttt{access}\ 3])]
\end{aligned}$$

If we load ops into an empty machine configuration we get the following computation.

$$\begin{aligned}
&((ops, [], [], []), []) \\
&\xrightarrow{\beta} (([\texttt{select}\, fst], [], [\iota_1], []), \Sigma_1) \quad \text{by } (\beta \text{ Object}) \\
&\qquad \text{where } \Sigma_1 = [\iota_1 \mapsto [(fst, ops_1), (snd, ops_2), (swap, ops_3)]] \\
&\xrightarrow{\beta} ((ops_1, [\iota_1], [], [([], [])]), \Sigma_1) \quad \text{by } (\beta \text{ Select}) \\
&\xrightarrow{\beta} (([], [\iota_1], [\iota_2], [([], [])]), \Sigma_2) \quad \text{by } (\beta \text{ Object}) \\
&\qquad \text{where } \Sigma_2 = (\iota_2 \mapsto []) :: \Sigma_1 \\
&\xrightarrow{\tau} (([], [], [\iota_2], []), \Sigma_2) \quad \text{by } (\tau \text{ Return})
\end{aligned}$$

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 terminal configuration $(([], [], [\iota_2], []), \Sigma_2)$. The location ι_2 returned on the argument stack references an empty object in the store.

Example 2: ZAM-style function call

As a second example, let the term $a = (\lambda(x)x)(\lambda(x)[])[]$. We have $[] \vdash a \Rightarrow ops$, where the operation list ops is given by:

$$\begin{aligned}
ops &= [\texttt{pushmark}, \texttt{object}[], \texttt{cur}\ ops_2, \texttt{cur}\ ops_1, \texttt{apply}] \\
ops_1 &= [\texttt{access}\ 1, \texttt{return}] \\
ops_2 &= [\texttt{object}[], \texttt{return}]
\end{aligned}$$

If we load ops into an empty machine configuration we get the following computation.

$$
\begin{array}{ll}
\multicolumn{2}{l}{((ops, [], [], []), [])} \\
\xrightarrow{\tau} & (([\mathtt{object}[], \mathtt{cur}\ ops_2, \mathtt{cur}\ ops_1, \mathtt{apply}], [], [\diamond], []), []) \\
& \qquad \text{by } (\tau \text{ Pushmark}) \\
\xrightarrow{\tau} & (([\mathtt{cur}\ ops_2, \mathtt{cur}\ ops_1, \mathtt{apply}], [], [\iota_1, \diamond], []), \Sigma_1 \overset{\text{def}}{=} [\iota_1 \mapsto []]) \\
& \qquad \text{by } (\beta \text{ Object}) \\
\xrightarrow{\tau} & (([\mathtt{cur}\ ops_1, \mathtt{apply}], [], [\mathit{fun}(ops_2, []), \iota_1, \diamond], []), \Sigma_1) \\
& \qquad \text{by } (\tau \text{ Cur}) \\
\xrightarrow{\tau} & (([\mathtt{apply}], [], [\mathit{fun}(ops_1, []), \mathit{fun}(ops_2, []), \iota_1, \diamond], []), \Sigma_1) \\
& \qquad \text{by } (\tau \text{ Cur}) \\
\xrightarrow{\beta} & ((ops_1, [\mathit{fun}(ops_2, [])], [\iota_1, \diamond], [([], [])]), \Sigma_1) \quad \text{by } (\beta \text{ Apply}) \\
\xrightarrow{\tau} & (([\mathtt{return}], [\mathit{fun}(ops_2, [])], [\mathit{fun}(ops_2, []), \iota_1, \diamond], [([], [])]), \Sigma_1) \\
& \qquad \text{by } (\tau \text{ Access}) \\
\xrightarrow{\beta} & ((ops_2, [\iota_1], [\diamond], [([], [])]), \Sigma_1) \quad \text{by } (\beta \text{ Function Return}) \\
\xrightarrow{\tau} & (([\mathtt{return}], [\iota_1], [\iota_2, \diamond], [([], [])]), \Sigma_2 \overset{\text{def}}{=} (\iota_2 \mapsto []) :: \Sigma_1) \\
& \qquad \text{by } (\beta \text{ Object}) \\
\xrightarrow{\tau} & (([], [], [\iota_2], []), \Sigma_2) \quad \text{by } (\tau \text{ Function Return})
\end{array}
$$

We see in this example the mechanism for function application, and in particular how, like the ZAM, our abstract machine uses a mark on the stack to delimit a series of arguments to a function.

The function call begins with the (τ Pushmark) τ-transition. The abstract machine evaluates applications in a right-to-left fashion, pushing the results of evaluating the arguments onto the argument stack. The closure representing the function to be called is pushed onto the argument stack, and the (β Apply) β-transition starts the body of the function $\lambda(x)x$ applied to the first argument and pushes an entry on the return stack. During the (β Function Return) β-transition, which does not touch the return stack, the outcome of this application gets applied to the second curried argument. The (τ Function Return) τ-transition completes the application by popping the entry off the return stack.

In the terminal configuration, $(([], [], [\iota_2], []), \Sigma_2)$ we have a location ι_2 on the argument stack. At location ι_2 in the store Σ_2 is an empty object $[]$. This evaluation produces some garbage in the store, at location ι_1.

Example 3: ZAM-style Curried function call

As a third example, let the term $a = (\lambda(xyz)x)[][]$. We have $[] \vdash a \Rightarrow ops$, where the operation list ops is given by:

$$
\begin{array}{rcl}
ops & = & [\mathtt{pushmark}, \mathtt{object}[], \mathtt{object}[], \mathtt{cur}(ops_1), \mathtt{apply}] \\
ops_1 & = & [\mathtt{grab}, \mathtt{grab}, \mathtt{access}\ 3, \mathtt{return}]
\end{array}
$$

If we load ops into an empty machine configuration we get the following computation.

$$
\begin{array}{ll}
((ops, [], [], []), []) & \\
\xrightarrow{\tau} & (([\texttt{object}[], \texttt{object}[], \texttt{cur}(ops_1), \texttt{apply}], [], [\diamond], []), []) \\
& \quad \text{by } (\tau \text{ Pushmark}) \\
\xrightarrow{\beta} & (([\texttt{object}[], \texttt{cur}(ops_1), \texttt{apply}], [], [\iota_1, \diamond], []), \Sigma_1) \quad \text{by } (\beta \text{ Object}) \\
& \quad \text{where } \Sigma_1 = [\iota_1 \mapsto []] \\
\xrightarrow{\beta} & (([\texttt{cur}(ops_1), \texttt{apply}], [], [\iota_2, \iota_1, \diamond], []), \Sigma_2) \quad \text{by } (\beta \text{ Object}) \\
& \quad \text{where } \Sigma_2 = [\iota_1 \mapsto [], \iota_2 \mapsto []] \\
\xrightarrow{\tau} & (([\texttt{apply}], [], [\mathit{fun}(ops_1, []), \iota_2, \iota_1, \diamond], []), \Sigma_2) \quad \text{by } (\tau \text{ Cur}) \\
\xrightarrow{\beta} & ((ops_1, [\iota_2], [\iota_1, \diamond], [([], [])]), \Sigma_2) \quad \text{by } (\beta \text{ Apply}) \\
\xrightarrow{\beta} & (([\texttt{grab}, \texttt{access } 3, \texttt{return}], [\iota_1, \iota_2], [\diamond], [([], [])]), \Sigma_2) \quad \text{by } (\beta \text{ Grab}) \\
\xrightarrow{\tau} & (([], [], [\mathit{fun}([\texttt{access } 3, \texttt{return}], [\iota_1, \iota_2])], []), \Sigma_2) \quad \text{by } (\tau \text{ Grab})
\end{array}
$$

Consider the transitions corresponding to the application of the function $\lambda(xyz)x$ to its two curried arguments $[]$ and $[]$. The curried call begins with the (τ Pushmark) τ-transition, which pushes a mark, $\diamond$, onto the argument stack. After the two arguments have been evaluated, the (β Apply) β-transition starts the body of the function $\lambda(xyz)x$ applied to the first curried argument, $[]$, and pushes an entry on the return stack. The (β Grab) β-transition applies the curried function $\lambda(yz)x$ to the second argument, $[]$. The second `grab` instruction finds a mark on the stack indicating there are no more arguments to be consumed, so causes a (τ Grab) τ-transition, which builds a closure and returns, popping an entry off the return stack.

The terminal configuration is:

$$(([], [], [\mathit{fun}([\texttt{access } 3, \texttt{return}], [\iota_1, \iota_2])], []), \Sigma_2)$$

We will show formally in Section 3.4 that the function closure returned on the argument stack, $\mathit{fun}([\texttt{access } 3, \texttt{return}], [\iota_1, \iota_2])$, represents the function $\lambda(z)\iota_2$.

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 that is a decompiled version of the original 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
$a^\diamond, b^\diamond ::= a \mid \diamond$	term or mark
$as ::= [a_i^\diamond{}^{\,i \in 1..n}]$	unloading stack

Next we make a simultaneous inductive definition of a *u-transition* relation $p \xrightarrow{u} p'$, and three unloading relations: $(ops, e) \leadsto (x)b$, that unloads a method closure to a method, $fun(ops, e) \leadsto \lambda(x)b$, that unloads a function closure to a λ-abstraction and $[U_i^{\diamond}\ ^{i\in 1..n}] \leadsto [a_i^{\diamond}\ ^{i\in 1..n}]$, 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 $(ops_i, e) \leadsto (x_i)b_i$ 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 $(ops, e) \leadsto (x)b$.
(u Let) $(\texttt{let}(ops') :: ops'', e, a :: as, RS) \xrightarrow{u} (ops'', e, (let\ \ x = a\ \ in\ \ b) :: as, RS)$
if $(ops', e) \leadsto (x)b$.
(u Return) $([], e, as, (ops, E) :: RS) \xrightarrow{u} (ops, e', as, RS)$
if $E \leadsto e'$.
(u Cur) $(\texttt{cur}\ ops :: ops', e, as, RS) \xrightarrow{u} (ops', e, (\lambda(x)a) :: as, RS)$
if $fun(ops, e) \leadsto \lambda(x)a$.
(u Function Return) $([\texttt{return}], e, [a_i\ ^{i\in 1..n}]@[\diamond]@as, RS) \xrightarrow{u}$
$([], e, (a_1(a_2)\cdots(a_n)) :: as, RS)$.
(u Grab) $(\texttt{grab} :: ops, e, as, RS) \xrightarrow{u} ([\texttt{return}], e, (\lambda(x)a) :: as, RS)$
if $fun(ops, e) \leadsto \lambda(x)a$.
(u Apply) $(\texttt{apply} :: ops, e, [a_i\ ^{i\in 1..n}]@[\diamond]@as, RS) \xrightarrow{u}$
$(ops, e, (a_1 a_2 \ldots a_n) :: as, RS)$.
(u Pushmark) $(\texttt{pushmark} :: ops, e, as, RS) \xrightarrow{u} (ops, e, \diamond :: as, RS)$.

(Unload Abstraction) $(ops, e) \leadsto (x)b$
if $x \notin fv(e)$ and $(ops, x :: e, [], []) \xrightarrow{u}{}^{*} ([], e', [b], [])$.
(Unload Closure) $fun(ops, e) \leadsto \lambda(x)b$
if $x \notin fv(e)$ and $(ops, x :: e, [\diamond], []) \xrightarrow{u}{}^{*} ([], e', [b], [])$.
(Unload List Empty) $[] \leadsto []$.
(Unload List Loc) $\iota :: [U_i^{\diamond}\ ^{i\in 1..n}] \leadsto \iota :: [a_i^{\diamond}\ ^{i\in 1..n}]$
if $[U_i^{\diamond}\ ^{i\in 1..n}] \leadsto [a_i^{\diamond}\ ^{i\in 1..n}]$.
(Unload List Closure) $fun(ops, E) :: [U_i^{\diamond}\ ^{i\in 1..n}] \leadsto (\lambda(x)a) :: [a_i^{\diamond}\ ^{i\in 1..n}]$
if $[U_i^{\diamond}\ ^{i\in 1..n}] \leadsto [a_i^{\diamond}\ ^{i\in 1..n}]$, $E \leadsto e$ and $fun(ops, e) \leadsto \lambda(x)a$.
(Unload List Mark) $\diamond :: [U_i^{\diamond}\ ^{i\in 1..n}] \leadsto \diamond :: [a_i^{\diamond}\ ^{i\in 1..n}]$
if $[U_i^{\diamond}\ ^{i\in 1..n}] \leadsto [a_i^{\diamond}\ ^{i\in 1..n}]$.

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

(Unload Object) $[(\ell_i, (ops_i, E_i))\ ^{i\in 1..n}] \leadsto [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$
if $E_i \leadsto e_i$ and $(ops_i, e_i) \leadsto (x_i)b_i$ for all $i \in 1..n$.
(Unload Store) $[\iota_i \mapsto O_i\ ^{i\in 1..n}] \leadsto [\iota_i \mapsto o_i\ ^{i\in 1..n}]$ if $O_i \leadsto o_i$ for all $i \in 1..n$.
(Unload Config) $((ops, E, AS, RS), \Sigma) \leadsto (a, \sigma)$
if $\Sigma \leadsto \sigma$, $E \leadsto e$, $AS \leadsto as$ and $(ops, e, as, RS) \xrightarrow{u}{}^{*} ([], e', [a], [])$.

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 $\Sigma \rightsquigarrow \sigma$.

Two important facts about the unloading relation $\rightsquigarrow$ are that no u transition can prevent unloading, and that the unloading relation $\rightsquigarrow$ is deterministic:

Lemma 10
Suppose $p \xrightarrow{u} q$. Then for all a, $p \rightsquigarrow a$ if and only if $q \rightsquigarrow a$.

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

Proposition 11
Whenever $p \rightsquigarrow a$ and $p \rightsquigarrow a'$, $a = a'$.

3.4 Examples of unloading

To clarify the workings of the unloading machine, we present some examples. We unload some of the abstract machine states of the examples in section 3.2.

Example 1: Unloading a compiled term

Recall from Example 3 of section 3.2 the configuration $((ops, [], [], []), [])$, where

$$\begin{aligned} ops &= [\texttt{pushmark}, \texttt{object}[], \texttt{object}[], \texttt{cur}(ops_1), \texttt{apply}] \\ ops_1 &= [\texttt{grab}, \texttt{grab}, \texttt{access}\ 3, \texttt{return}] \end{aligned}$$

We know already that $[] \vdash (\lambda(xyz)x)[][] \Rightarrow ops$.

We aim to prove $((ops, [], [], []), []) \rightsquigarrow ((\lambda(xyz)x)[][], [])$. We build up to this result in four steps. The first step corresponds to unloading the body of the function $\lambda(xyz)x$ and each subsequent step will build a function whose body is the result of the previous step. Bound names are lost in translation, but since we identify terms up to alpha conversion, we choose variables in this example so that the unloaded term is the same as the original term.

(1) We compute:

$$\begin{aligned} &([\texttt{access}\ 3, \texttt{return}], [z, y, x], [\diamond], []) \\ &\quad \xrightarrow{u} ([\texttt{return}], [z, y, x], [x, \diamond], []) \quad \text{by } (u \text{ Access}) \\ &\quad \xrightarrow{u} ([], [z, y, x], [x], []) \quad \text{by } (u \text{ Function Return}) \end{aligned}$$

By rule (Unload Closure), we get:

$$fun([\texttt{access}\ 3, \texttt{return}], [y, x]) \rightsquigarrow \lambda(z)x$$

(2) Hence, we compute:

$$\begin{aligned} &([\texttt{grab}, \texttt{access}\ 3, \texttt{return}], [y, x], [\diamond], []) \\ &\quad \xrightarrow{u} ([\texttt{return}], [y, x], [\lambda(z)x, \diamond], []) \quad \text{by } (u \text{ Grab}) \\ &\quad \xrightarrow{u} ([], [y, x], [\lambda(z)x], []) \quad \text{by } (u \text{ Function Return}) \end{aligned}$$

By (Unload Closure), we get:

$$\mathit{fun}([\texttt{grab}, \texttt{access }3, \texttt{return}], [x]) \rightsquigarrow \lambda(yz)x$$

(3) Hence, we compute:

$$\begin{aligned}&(\mathit{ops}_1, [x], [\diamond], [])\\ &\xrightarrow{u} ([\texttt{return}], [x], [(\lambda(yz)x), \diamond], []) \quad \text{by } (u \text{ Grab})\\ &\xrightarrow{u} ([], [x], [\lambda(yz)x], []) \quad \text{by } (u \text{ Function Return})\end{aligned}$$

Again by (Unload Closure), we get:

$$\mathit{fun}([\texttt{grab}, \texttt{grab}, \texttt{access }3, \texttt{return}], []) \rightsquigarrow \lambda(xyz)x$$

(4) Below, the result of step (3) is used in the (u Cur) step:

$$\begin{aligned}&(\mathit{ops}, [], [], [])\\ &\xrightarrow{u} ([\texttt{object}[], \texttt{object}[], \texttt{cur}(\mathit{ops}_1), \texttt{apply}], [], [\diamond], [])\\ &\qquad \text{by } (u \text{ Pushmark})\\ &\xrightarrow{u} ([\texttt{object}[], \texttt{cur}(\mathit{ops}_1), \texttt{apply}], [], [[], \diamond], []) \quad \text{by } (u \text{ Object})\\ &\xrightarrow{u} ([\texttt{cur}(\mathit{ops}_1), \texttt{apply}], [], [[], [], \diamond], []) \quad \text{by } (u \text{ Object})\\ &\xrightarrow{u} ([\texttt{apply}], [], [(\lambda(xyz)x), [], [], \diamond], []) \quad \text{by } (u \text{ Cur})\\ &\xrightarrow{u} ([], [], [(\lambda(xyz)x)[][]], []) \quad \text{by } (u \text{ Apply})\end{aligned}$$

The terminal configuration of the unloading machine has our original expression $(\lambda(xyz)x)[][]$ on the stack. Hence by (Unload Config) we have $((\mathit{ops}, [], [], []), []) \rightsquigarrow ((\lambda(xyz)x)[][], [])$ as desired.

Example 2: Unloading a terminal configuration

For the next example, we unload the terminal configuration of Example 3 of section 3.2, $(([], [], [\mathit{fun}([\texttt{access }3, \texttt{return}], [\iota_1, \iota_2])], []), \Sigma_2)$, where $\Sigma_2 = [\iota_1 \mapsto [], \iota_2 \mapsto []]$.

From rule (Unload Store) we have $\Sigma_2 \rightsquigarrow \sigma_2 = [\iota_1 \mapsto [], \iota_2 \mapsto []]$. To unload the closure $\mathit{fun}([\texttt{access }3, \texttt{return}], [\iota_1, \iota_2])$, we calculate:

$$\begin{aligned}&([\texttt{access }3, \texttt{return}], [z, \iota_1, \iota_2], [\diamond], [])\\ &\xrightarrow{u} ([\texttt{return}], [z, \iota_1, \iota_2], [\iota_2, \diamond], []) \quad \text{by } (u \text{ Access})\\ &\xrightarrow{u} ([], [z, \iota_1, \iota_2], [\iota_2], []) \quad \text{by } (u \text{ Function Return})\end{aligned}$$

By rule (Unload Closure) we get:

$$\mathit{fun}([\texttt{access }3, \texttt{return}], [\iota_1, \iota_2]) \rightsquigarrow \lambda(z)\iota_2$$

From rules (Unload List Closure) and (Unload List Empty) we get that the argument stack unloads as follows:

$$[\mathit{fun}([\texttt{access }3, \texttt{return}], [\iota_1, \iota_2])] \rightsquigarrow [\lambda(z)\iota_2]$$

Finally, by (Unload Config) we deduce:

$$(([], [], [\mathit{fun}([\texttt{access }3, \texttt{return}], [\iota_1, \iota_2])], []), \Sigma_2) \rightsquigarrow (\lambda(z)\iota_2, \sigma_2)$$

Combining the working from this section and section 3.2, we have shown that unloading the outcome of compiling and executing the term $(\lambda(xyz)x)[][]$, yields the configuration $(\lambda(z)\iota_2, [\iota_1 \mapsto [], \iota_2 \mapsto []])$.

Example 3: Unloading an intermediate configuration

For a final example, we consider an intermediate configuration obtained from the evaluation of $(\lambda(x)x.\ell)[\ell = \varsigma(s)\lambda(y)y][]$ in the abstract machine. The configuration we will unload is:

$$(([\texttt{select}\ \ell, \texttt{return}], [\iota_2], [\iota_2, \iota_1, \diamond], [([], [])]), \Sigma_2)$$

where

$$\Sigma_2 = [\iota_1 \mapsto [], \iota_2 \mapsto [(\ell, ([\texttt{cur}([\texttt{access}\ 1, \texttt{return}])], []))]]$$

We first unload the store:

- We compute:

$$\begin{array}{l} ([\texttt{access}\ 1, \texttt{return}], [y, s], [\diamond], []) \\ \quad \xrightarrow{u} \quad ([\texttt{return}], [y, s], [y, \diamond], []) \quad \text{by } (u \text{ Access}) \\ \quad \xrightarrow{u} \quad ([], [y, s], [y], []) \quad \text{by } (u \text{ Function Return}) \end{array}$$

  So by rule (Unload Closure), $\mathit{fun}([\texttt{access}\ 1, \texttt{return}], [s]) \leadsto \lambda(y)y$.
- Hence, we get:

$$([\texttt{cur}([\texttt{access}\ 1, \texttt{return}])], [s], [], []) \xrightarrow{u} ([], [s], [\lambda(y)y], [])$$

  By (Unload Abstraction) we get:

$$([\texttt{cur}([\texttt{access}\ 1, \texttt{return}])], []) \leadsto (s)\lambda(y)y$$

- Hence by rule (Unload Store):

$$\Sigma_2 \leadsto [\iota_1 \mapsto [], \iota_2 \mapsto [\ell = \varsigma(s)\lambda(y)y]]$$

To unload the other component of the configuration, we compute:

$$\begin{array}{l} ([\texttt{select}\ \ell, \texttt{return}], [\iota_2], [\iota_2, \iota_1, \diamond], [([], [])]) \\ \quad \xrightarrow{u} \quad ([\texttt{return}], [\iota_2], [\iota_2.\ell, \iota_1, \diamond], [([], [])]) \quad \text{by } (u \text{ Select}) \\ \quad \xrightarrow{u} \quad ([], [\iota_2], [(\iota_2.\ell)\iota_1], [([], [])]) \quad \text{by } (u \text{ Function Return}) \\ \quad \xrightarrow{u} \quad ([], [], [(\iota_2.\ell)\iota_1], []) \quad \text{by } (u \text{ Return}) \end{array}$$

By rule (Unload Config) we deduce:

$$\begin{array}{l} (([\texttt{select}\ \ell, \texttt{return}], [\iota_2], [\iota_2, \iota_1, \diamond], [([], [])]), \Sigma_2) \\ \quad \leadsto \quad ((\iota_2.\ell)\iota_1, [\iota_1 \mapsto [], \iota_2 \mapsto [\ell = \varsigma(s)\lambda(y)y]]) \end{array}$$

3.5 Correctness of the abstract machine

We first show that unloading is an inverse to compilation:

Proposition 12
Whenever $[] \vdash a \Rightarrow ops$ then $((ops, [], [], []), []) \rightsquigarrow (a, [])$.

Proof
We prove a more general fact: if $x_i{}^{i\in 1..n} \vdash a \Rightarrow ops$ then for all $b_i{}^{i\in 1..n}$

$$(ops, [b_i{}^{i\in 1..n}], [], []) \xrightarrow{u}{}^{*} ([], [b_i{}^{i\in 1..n}], [a\{\!\!\{b_i/x_i{}^{i\in 1..n}\}\!\!\}], [])$$

by induction on the derivation of $x_i{}^{i\in 1..n} \vdash a \Rightarrow ops$. $\square$

The next lemma asserts 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, functions or marks ($\diamondsuit$).

Lemma 13
If $(ops, e, as, RS) \xrightarrow{u} (ops', e', as', RS')$ and $as = [a_i^{\diamondsuit}{}^{i\in 1..n}, \mathscr{R}, u_j^{\diamondsuit}{}^{i\in 1..m}]$ where $\bullet \notin fv(e)$ then $\bullet \notin fv(e')$ and $as' = [b_i^{\diamondsuit}{}^{i\in 1..n'}, \mathscr{R}', v_j^{\diamondsuit}{}^{j\in 1..m'}]$ for some $\mathscr{R}'$, $b_i^{\diamondsuit}$ and $v_j^{\diamondsuit}$ (with $i \in 1..n', j \in 1..m'$).

The unloading machine also preserves substitutions:

Lemma 14
If $p \xrightarrow{u} q$ then $p\{\!\!\{a/x\}\!\!\} \xrightarrow{u} q\{\!\!\{a/x\}\!\!\}$.

The next lemma shows that the unloading machine is independent of the terms in its environment and on its stack. 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.

Lemma 15
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
By induction on the derivation of $p \xrightarrow{u} p'$. $\square$

A corollary of Lemma 15 is the following:

Lemma 16
If $p \rightsquigarrow a$ then for all q with p and q shape-mark-equivalent, there is an a' with $q \rightsquigarrow a'$.

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

Proposition 17
Whenever $(ops, e, a :: [u_i^{\diamondsuit}{}^{i\in 1..n}], RS) \rightsquigarrow b$, where $\bullet \notin fv(e)$, there is a reduction context, $\mathscr{R}$, such that $(ops, e, a' :: [u_i^{\diamondsuit}{}^{i\in 1..n}], RS) \rightsquigarrow \mathscr{R}[a']$ for any a'.

Proof

If $(ops, e, a :: [u_i^{\diamond}\ ^{i\in 1..n}], RS) \rightsquigarrow b$ there is a b' such that $(ops, e, \bullet :: [u_i^{\diamond}\ ^{i\in 1..n}], RS) \rightsquigarrow b'$ (by Lemma 16). This means $(ops, e, \bullet :: [u_i^{\diamond}\ ^{i\in 1..n}], RS) \xrightarrow{u}{}^{k} ([], e', [b'], [])$ for some k. Since $\bullet$ is a reduction context, applying Lemma 13 k times tells us that $b' = \mathscr{R}$ for some $\mathscr{R}$. Since $\bullet \notin fv(e)$, Lemma 14 implies $(ops, e, a' :: [u_i^{\diamond}\ ^{i\in 1..n}], RS) \rightsquigarrow \mathscr{R}[a']$ (because $a' = \bullet\{\!\!\{a'/\!\bullet\}\!\!\}$ and $\mathscr{R}[a'] = \mathscr{R}\{\!\!\{a'/\!\bullet\}\!\!\}$). $\square$

The first main lemma asserts that β transitions of the abstract machine correspond to reductions in our extended object calculus, and that τ transitions are not reflected in the source level reductions:

Lemma 18

(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 $c \to d$.

Proof

(1) The proof for each of the τ transitions 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 (Unload Config)). By (u Access), $p \xrightarrow{u} q$, so by Lemma 10 and $p \rightsquigarrow a$ we have $q \rightsquigarrow a$. So $D \rightsquigarrow (a, \sigma)$ as required.

(2) We examine each rule that may derive $C \xrightarrow{\beta} D$. We detail one case for illustration.

(β Clone) Here $C = (P, \Sigma)$, where $P = (\texttt{clone} :: ops, E, \iota :: AS, RS)$, 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 $P \downarrow p = (\texttt{clone} :: ops, e, \iota :: as, RS)$, $E \rightsquigarrow e$, $AS \rightsquigarrow as$, $p \rightsquigarrow a$ and $\Sigma \rightsquigarrow \sigma$. By ($u$ Clone), $p \xrightarrow{u} (ops, e, (clone(\iota)) :: as, RS)$. Hence by Lemma 10, $(ops, e, (clone(\iota)) :: as, RS) \rightsquigarrow a$. Therefore by Proposition 17, there is a reduction context $\mathscr{R}$ such that for all a', $(ops, e, a' :: as, RS) \rightsquigarrow \mathscr{R}[a']$; by Proposition 11, $a = \mathscr{R}[clone(\iota)]$ and $q = (ops, e, \iota' :: as, RS) \rightsquigarrow \mathscr{R}[\iota']$. Let $\sigma' = (\iota' \mapsto \sigma(\iota)) :: \sigma$ so that $\Sigma' \rightsquigarrow \sigma'$ by (Unload Store). Let $d = (\mathscr{R}[\iota'], \sigma')$. $Q \downarrow q \rightsquigarrow \mathscr{R}[\iota']$, so $D = (Q, \Sigma') \rightsquigarrow d$. Finally, we have $c \to d$ by (Red Clone). $\square$

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, and that if the abstract machine is stuck then so is its unloaded source term.

Lemma 19

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 $f : (ops, E, AS, RS) \mapsto (|RS|, |ops|)$ 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 $C_1 \xrightarrow{\tau} C_2 \xrightarrow{\tau} \dots$ would give an infinite descending chain in $\mathbb{N} \times \mathbb{N}$, a contradiction since the lexicographic ordering is a well-ordering. $\square$

Lemma 20
If $C \rightsquigarrow c$ and there is no D with $C \xrightarrow{\beta\tau} D$ then there is no d with $c \to d$.

Proof
Let $C = (P, \Sigma)$, where $P = (ops, E, AS, RS)$. Now, $C \rightsquigarrow c$ means $P \downarrow p$, $\Sigma \rightsquigarrow \sigma$, $p \xrightarrow{u}{}^{*} ([], e', [a], [])$ (for some e'), and $c = (a, \sigma)$.

For a contradiction, suppose that there is no D such that $C \xrightarrow{\beta\tau} D$, but there is d such that $c \to d$. Given that $p \xrightarrow{u}{}^{*} ([], e', [a], [])$, either (1) $p = ([], e', [a], [])$ or (2) there is p' such that $p \xrightarrow{u} p'$ and $p' \xrightarrow{u}{}^{*} ([], e', [a], [])$.

In case (1), a must either be a function or a location, from the definition of $AS \rightsquigarrow as$ which forms part of the $P \downarrow p$ judgment. Then $c = (a, \sigma)$ is a value, so there is no d with $c \to d$.

In case (2), we consider two of the rules capable of deriving $p \xrightarrow{u} p'$. The cases for the other rules are similar.

(u Access) Here $p = (\texttt{access}\ j :: ops, e, as, RS)$ and $p' = (ops, e, u_j :: as, RS)$ where $e = [u_i{}^{i \in 1..n}]$ and $j \in 1..n$. Now, $P \downarrow p$ means $P = (\texttt{access}\ j :: ops, [U_i{}^{i \in 1..n}], AS, RS)$, $U_i \rightsquigarrow u_i$ for $i \in 1..n$ and $AS \rightsquigarrow as$. But then $C = (P, \Sigma) \xrightarrow{\tau} ((ops, [U_i{}^{i \in 1..n}], U_j :: AS, RS), \Sigma)$ by rule (τ Access) contradicting the non-existence of D with $C \xrightarrow{\beta\tau} D$.

(u Select) Here $p = (\texttt{select}\ \ell :: ops, e, u :: as', RS)$ and $p' = (ops, e, (u.\ell) :: as', RS)$. Now, $p' \to^{*} ([], e', [a], [])$ means $p' \rightsquigarrow a$. From $P \downarrow p$, we deduce $E \rightsquigarrow e$. We note that none of the unloading rules introduces a free variable without binding it, so $fv(e) = \varnothing$; in particular this implies $\bullet \notin fv(e)$. Hence we may apply Proposition 17 to $p' = (ops, e, (u.\ell) :: as', RS)$ to infer the existence of a reduction context $\mathscr{R}$ such that $p' \rightsquigarrow \mathscr{R}[u.\ell]$. Lemma 10 with $p' \rightsquigarrow \mathscr{R}[u.\ell]$ and $p' \rightsquigarrow a$ implies $a = \mathscr{R}[u.\ell]$ and $c = (\mathscr{R}[u.\ell], \sigma)$. If $c \to d$ then the only rule that can apply is (Red Select); hence $u = \iota$ and $\sigma(\iota) = o@[\ell = \varsigma(x)b]@o'$. From $P \downarrow p$ we derive $AS \rightsquigarrow \iota :: as'$ and $E \rightsquigarrow e$. From $AS \rightsquigarrow \iota :: as'$ and (Unload List Loc) we see that $AS = \iota :: AS'$ where $AS' \rightsquigarrow as'$. From $\Sigma \rightsquigarrow \sigma$, $\sigma(\iota) = o@[\ell = \varsigma(x)b]@o'$, (Unload Store) and (Unload Object) we deduce $\Sigma(\iota) = O@[\ell = (ops', E'')]@O'$ where $E'' \rightsquigarrow e''$ and $(ops', e'') \rightsquigarrow (x)b$. Hence $C = ((\texttt{select}\ \ell :: ops, E, \iota :: AS', RS), \Sigma)$. Finally, by rule ($\beta$ Select), we may derive $C \xrightarrow{\beta} ((ops', \iota :: E'', AS', RS), \Sigma)$ and hence a contradiction. $\square$

We are now in a position to show the second main lemma. It asserts that the abstract machine semantics simulates the semantics of the object calculus:

Lemma 21
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 19 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 20 there is no d with $c \to d$, contradicting the assumption of this lemma. So D' can make a β-transition. We consider each of the β-transition rules in turn.

(β Select) Here $D' = ((\texttt{select}\ \ell :: ops, E, \iota :: AS, RS), \Sigma)$ where $\Sigma(\iota) = O$ @ $[(\ell, (ops', E'))]$ @ O'. Moreover, $D' \downarrow (p, \sigma)$ where $p = (\texttt{select}\ \ell :: ops, e, \iota :: as, RS)$, $E \rightsquigarrow e$, $AS \rightsquigarrow as$, $\Sigma \rightsquigarrow \sigma$. Then $p \xrightarrow{u} (ops, e, (\iota.\ell) :: as, RS)$, and by Proposition 17 there is a reduction context $\mathscr{R}$ such that $p \rightsquigarrow \mathscr{R}[\iota.\ell]$. Hence, $c = (\mathscr{R}[\iota.\ell], \sigma)$ and if $c \to d'$ then $d = d'$, since (Red Select) is the unique rule which can derive $c \to d'$ and gives a unique 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 17, $D' \rightsquigarrow c = (\mathscr{R}[clone(\iota)], \sigma)$ where $\sigma(\iota) = o$ and $O \rightsquigarrow o$. Now $d = (\mathscr{R}[\iota'], \sigma + (\iota' \mapsto o))$ 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 17 again, we get $D \rightsquigarrow (\mathscr{R}[\iota'], \sigma + (\iota' \mapsto o)) = d$ as required.

(β Object) Similar to (β Clone). $\square$

We call a configuration of the form $(([], E, [V], []), \Sigma)$ *terminal.*

Lemma 22
If $C \rightsquigarrow c$ and not $C \xrightarrow{\tau}$ then C is terminal if and only if c is terminal.

It follows from Lemmas 18, 21 and 22 that the semantics of the abstract machine and that of our extended object calculus are related via the unloading relation. Let $C \searrow D$ if $C \xrightarrow{\beta\tau}^* D$ and D is terminal.

Lemma 23
(1) If $C \rightsquigarrow c$ and $C \searrow D$ then there is a d with $D \rightsquigarrow d$ and $c \searrow d$.
(2) If $C \rightsquigarrow c$ and $c \searrow d$ then there is a D with $D \rightsquigarrow d$ and $C \searrow D$.

We are now in a position to prove the main result:

Theorem 3
Suppose that $[] \vdash a \Rightarrow ops$. Then, for all d, $(a, []) \searrow d$ if and only if there is a D with $((ops, [], [], []), []) \searrow D$ and $D \rightsquigarrow d$.

Proof
Given $[] \vdash a \Rightarrow ops$, Proposition 12 implies that $((ops, [], [], []), []) \rightsquigarrow (a, [])$. Suppose $(a, []) \searrow d$. By Lemma 23(2), there is D such that $D \rightsquigarrow d$ and $((ops, [], [], []), []) \searrow D$. Conversely, suppose there is D with $((ops, [], [], []), []) \searrow D$ and $D \rightsquigarrow d$. By Lemma 23(1), there is d' such that $D \rightsquigarrow d'$ and $(a, []) \searrow d'$. A corollary of Proposition 11 is that $D \rightsquigarrow d$ and $D \rightsquigarrow d'$ imply that $d = d'$. Therefore, we have $(a, []) \searrow d$, as desired. $\square$

3.6 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 ZAM's 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 and Miller, 1992; Plotkin, 1975; Sestoft, 1997). 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 *et al.*, 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 *et al.*, 1998) 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. Rittri uses a generic framework based on bisimulation to prove correctness of both a machine for evaluating arithmetic expressions, and the SECD machine. Our work goes beyond Rittri's by dealing with state and objects. We found it simpler to write a direct proof than to appeal to his generic framework.

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 bytecode arrays indexed by a program counter. Another difference is that our model omits garbage collection, which is essential to the implementation. Therefore Theorem 3 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 imperative object calculus. We consider only the core object calculus, not the calculus extended with functions. 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 (Closed Instances of Use) equivalence. Informally, 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 and locations are substituted for the free variables. 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, Theorem 4.

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 $(a, \sigma) \sim_w (a', \sigma')$, if and only if $\vdash_w (a, \sigma)\ ok$, $\vdash_w (a', \sigma')\ ok$ and, for all reduction contexts with $locs(\mathscr{R}) \subseteq w$ and $fv(\mathscr{R}) = \{\bullet\}$, $(\mathscr{R}[a], \sigma) \updownarrow (\mathscr{R}[a'], \sigma')$.

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

The index set w is a view into the configurations: the locations in the stores that $\mathscr{R}$ 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, that is, $c \sim_w c'$ holds whenever $c \sim_{w'} c'$ and $w \subseteq w'$.

We can prove that reduction is sound with respect to experimental equivalence:

Lemma 24
If $\vdash_w c\ ok$ and $c \to c'$, then $c \sim_w c'$.

Proof
Suppose $\vdash_w (a, \sigma)\ ok$ and $(a, \sigma) \to (a', \sigma')$. Then $\vdash_w (a', \sigma')\ ok$ holds by Lemma 1.

Further, suppose $locs(\mathscr{R}) \subseteq w$ and $fv(\mathscr{R}) = \{\bullet\}$. By inspection of the reduction rules we see that $(a, \sigma) \to (a', \sigma')$ implies $(\mathscr{R}[a], \sigma) \to (\mathscr{R}[a'], \sigma')$. Clearly, $(\mathscr{R}[a'], \sigma')\Downarrow$ implies $(\mathscr{R}[a], \sigma)\Downarrow$ because any converging reduction sequence from $(\mathscr{R}[a'], \sigma')$ extends to a converging reduction sequence from $(\mathscr{R}[a], \sigma)$. The reverse implication follows because reduction is deterministic up to structural equivalence at w, that is, by a combination of Proposition 2 and Lemma 3. We conclude $(a, \sigma) \sim_w (a', \sigma')$, as required. $\square$

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

Lemma 25
Suppose $\vdash_w c$ *ok* and $\vdash_w c'$ *ok*. Then $c \sim_w c'$ 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 $d \sim_w d'$, or
(2) neither c nor c' converges.

It is possible to formulate garbage collection principles for unused objects in terms of experimental equivalences. We call a location ι garbage in $(a, \sigma@[\iota \mapsto o]@\sigma')$ if the configuration is well formed, $\vdash (a, \sigma@[\iota \mapsto o]@\sigma')$ *ok*, and it is also well formed without $(\iota \mapsto o)$ in the store, $\vdash (a, \sigma@\sigma')$ *ok*; that is, a and $\sigma@\sigma'$ make no reference to ι. Reduction is independent of garbage:

Lemma 26
Suppose ι is *garbage* in $(a, \sigma@[\iota \mapsto o]@\sigma')$. Then $(a, \sigma@[\iota \mapsto o]@\sigma') \to^n (v, \sigma_n@[\iota \mapsto o_n]@\sigma'_n)$ if and only if $o = o_n$, $\iota \notin dom(\sigma_n@\sigma'_n)$, and $(a, \sigma@\sigma') \to^n (v, \sigma_n@\sigma'_n)$.

Proof
By induction on the length of the computations. $\square$

The lemma can be used to prove the following garbage collection law which says that if ι is garbage in a configuration c, it can be garbage collected up to experimental equivalence at any w such that $\vdash_w c$ *ok* and $\iota \notin w$.

Lemma 27
Suppose ι is *garbage* in $(a, \sigma@[\iota \mapsto o]@\sigma')$. If $\vdash_w (a, \sigma@\sigma')$ *ok* then we have $(a, \sigma@[\iota \mapsto o]@\sigma') \sim_w (a, \sigma@\sigma')$.

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 24, 25 and 27, 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 $\rho : \{x_1, \ldots, x_n\} \to w$ whenever $\rho = [x_i \mapsto \imath_i{}^{i \in 1..n}]$ and $\imath_i \in w$ 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.) Now, we define two static terms a and a' to be operationally equivalent, written $a \approx a'$, if and only if $(a\rho, \sigma) \sim_{dom(\sigma)} (a'\rho, \sigma)$ holds for all well formed stores σ and substitutions $\rho : fv(a) \cup fv(a') \to dom(\sigma)$.

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 of 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 $\mathscr{R}$) either both converge or neither converges.

Lemma 28
For all static terms a and a', $a \approx a'$ if and only if $(\mathscr{R}[a]\rho, \sigma) \updownarrow (\mathscr{R}[a']\rho, \sigma)$, for all static reduction contexts $\mathscr{R}$, well formed stores σ, and substitutions $\rho : fv(\mathscr{R}[a]) \cup fv(\mathscr{R}[a']) \to dom(\sigma)$.

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

An easy consequence of Lemma 28 is that operational equivalence is preserved by static reduction contexts:

Lemma 29
If $a \approx a'$ then $\mathscr{R}[a] \approx \mathscr{R}[a']$, for all static reduction contexts $\mathscr{R}$.

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

Lemma 30
If $\mathscr{R}[x] \approx \mathscr{R}'[x]$ and $x \notin fv(\mathscr{R}) \cup fv(\mathscr{R}')$, then $\mathscr{R}[a] \approx \mathscr{R}'[a]$, for all static terms a.

Proof
After expanding the definition of $\approx$, the proof proceeds by induction on the length of computations, using Lemma 25. $\square$

4.3 Laws of operational equivalence

From Lemma 30 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 now show a selection of such laws 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), presented here in the form given by Talcott (1998):

Proposition 31

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

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

Proof

Part (1) is immediate from definition of $\approx$ and Lemma 24. For (2), by Lemma 30 it suffices to show $(let\ x = x\ in\ \mathscr{R}[x]) \approx \mathscr{R}[x]$ which is immediate from (1). □

Moggi's eta law is just Proposition 31(2) with $\mathscr{R} = \bullet$. To prove associativity:

$$let\ x = a\ in\ (let\ x = a'\ in\ b) \quad \approx \quad let\ x = (let\ x = a\ in\ a')\ in\ b \tag{1}$$

we first use Proposition 31(1), Lemma 29 and Lemma 30 to rewrite the left hand side to

$$let\ x = a\ in\ (let\ x = (let\ x = x\ in\ a')\ in\ b)$$

which, by Proposition 31(2) with $\mathscr{R} = (let\ x = (let\ x = \bullet\ in\ a')\ in\ b)$, rewrites to the right hand side of (1).

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

Proposition 32

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

(1) $(let\ x_j = o\ in\ \mathscr{R}[x_j.\ell_j]) \approx (let\ x_j = o\ in\ \mathscr{R}[b_j])$

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

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

(4) $clone(o) \approx o$

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

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

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

Proof

Parts (1), (3) and (5) follow from definition of $\approx$ and a few applications of Lemma 24.

Part (2) is immediate from (1) and Proposition 31(2).

Part (4) follows from Proposition 31(2), (5) and (6):

$$clone(o) \ \approx\ (let\ x = o\ in\ clone(x)) \ \approx\ (let\ x = o\ in\ o) \ \approx\ o$$

where $x \notin fv(o)$.

Part (6) is direct from the definition of $\approx$, Lemma 24 and Lemma 27.

Part (7) requires a more elaborate argument, first expanding the definition of $\approx$ and then analysing the possible reduction sequences of arbitrary closed instances, exploiting that reduction is independent of garbage, Lemma 26. $\square$

The next proposition gives laws for method update and its interaction with method selection and cloning.

Proposition 33

Let notation $a;b$ abbreviate $let\ \ x = a\ \ in\ \ b$ where $x \notin fv(b)$.

(1) $(let\ \ x = a.\ell \Leftarrow \varsigma(x)b\ \ in\ \ \mathscr{R}[x.\ell]) \approx (let\ \ x = a.\ell \Leftarrow \varsigma(x)b\ \ in\ \ \mathscr{R}[b])$

(2) $(let\ \ x = a.\ell \Leftarrow \varsigma(x)b\ \ in\ \ \mathscr{R}[x]) \approx (let\ \ x = a\ \ in\ \ \mathscr{R}[x.\ell \Leftarrow \varsigma(x)b])$

(3) $(a.\ell \Leftarrow \varsigma(x)b).\ell \Leftarrow \varsigma(x')b' \approx a.\ell \Leftarrow \varsigma(x')b'$

(4) $(y.\ell \Leftarrow \varsigma(x)b);(z.\ell' \Leftarrow \varsigma(x')b');a \approx (z.\ell' \Leftarrow \varsigma(x')b');(y.\ell \Leftarrow \varsigma(x)b);a$, if $\ell \neq \ell'$

(5) $clone(y.\ell \Leftarrow \varsigma(x)b) \approx (let\ \ z = clone(y)\ \ in\ \ (y.\ell \Leftarrow \varsigma(x)b);z.\ell \Leftarrow \varsigma(x)b)$

Proof

We prove only (1). The other laws are proved similarly. By Lemma 30 it suffices to show the law for some $y \notin fv(b)$ in place of a. This case holds by definition of $\approx$ and, if y is instantiated to a location pointing to an object with an ℓ method, by five applications of Lemma 24; if the object has no ℓ method, neither side of the equation converges. $\square$

Let us look at two examples of equational reasoning using the laws above.

Example 1: Pairs

Recall that $pair(a,b)$ is the object:

$$[fst = \varsigma(s)a, snd = \varsigma(s)b, swap = \varsigma(s)let\ \ x = s.fst\ \ in\ \ let\ \ y = s.snd\ \ in \\ (s.fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')x]$$

for some $s \notin fv(a) \cup fv(b)$. First, let us prove that the fst and snd methods work as projections:

$$\begin{array}{rcll} pair(a,b).fst & \approx & let\ \ s = pair(a,b)\ \ in\ \ a & \text{by Prop. 32(2)} \\ & \approx & a & \text{by Prop. 32(6)} \end{array}$$

Analogously, we derive that $pair(a,b).snd \approx b$.

To show that the *swap* method indeed swaps the components of a pair, we can argue as follows:

$$\begin{array}{lll}
\multicolumn{2}{l}{pair(x,y).swap} & \\
\approx & let\ \ s = pair(x,y)\ \ in & \\
 & \quad let\ \ x' = s.fst\ \ in\ \ let\ \ y' = s.snd\ \ in & \\
 & \qquad (s.fst \Leftarrow \varsigma(s')y').snd \Leftarrow \varsigma(s')x' & \text{by Prop. 32(2)} \\
\approx & let\ \ s = pair(x,y)\ \ in & \\
 & \quad let\ \ x' = x\ \ in\ \ let\ \ y' = s.snd\ \ in & \\
 & \qquad (s.fst \Leftarrow \varsigma(s')y').snd \Leftarrow \varsigma(s')x' & \text{by Prop. 32(1)} \\
\approx & let\ \ s = pair(x,y)\ \ in & \\
 & \quad let\ \ y' = s.snd\ \ in & \\
 & \qquad (s.fst \Leftarrow \varsigma(s')y').snd \Leftarrow \varsigma(s')x & \text{by Prop. 32(7) and 31(1)} \\
\approx & let\ \ s = pair(x,y)\ \ in & \\
 & \quad let\ \ y' = y\ \ in & \\
 & \qquad (s.fst \Leftarrow \varsigma(s')y').snd \Leftarrow \varsigma(s')x & \text{by Prop. 32(1)} \\
\approx & let\ \ s = pair(x,y)\ \ in & \\
 & \quad (s.fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')x & \text{by Prop. 32(7) and 31(1)} \\
\approx & (pair(x,y).fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')x & \text{by Prop. 31(2)} \\
\approx & pair(y,y).snd \Leftarrow \varsigma(s')x & \text{by Prop. 32(3)} \\
\approx & pair(y,x) & \text{by Prop. 32(3)}
\end{array}$$

We note that $pair(a,b).swap \approx pair(b,a)$ fails in general, for instance if a or b diverges, because a and b are evaluated in the course of the swap on the left hand side and they are not evaluated on the right hand side. However, by an elaboration of the previous derivation, we can show:

$$pair(a,b).swap \ \approx\ let\ \ x = a\ \ in\ \ let\ \ y = b\ \ in\ \ pair(y,x)$$

for arbitrary static terms a and b with $x \notin fv(b)$.

Example 2: Functions

For the second example, recall the encoding of call-by-value functions from Section 2.1:

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

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:

$$(\lambda(x)b)(y) \quad \approx \quad b\{\!\{y/x\}\!\} \qquad (2)$$

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

$(\lambda(x)b)(y)$

$\approx$	$((\lambda(x)b).arg \Leftarrow \varsigma(z)y).val$	by Prop. 31(1)
$\approx$	$o.val$	by Prop. 32(3) and Lemma 29
$\approx$	$let\ s = o\ in\ let\ x = s.arg\ in\ b$	by Prop. 32(2)
$\approx$	$let\ x = o.arg\ in\ b$	by Prop. 31(2)
$\approx$	$let\ x = (let\ z = o\ in\ y)\ in\ b$	by Prop. 32(2) and Lemma 29
$\approx$	$let\ x = y\ in\ b$	by Prop. 32(6) and Lemma 29
$\approx$	$b\{\!\{y/x\}\!\}$	by Prop. 31(1)

These examples 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 (2) used the fact that operational equivalence is preserved by reduction contexts, Lemma 29. 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 constructors and with immediate subterms pairwise related by $\mathcal{S}$, as defined by the following axiom schemes.

(Comp x**)** $x\, \widehat{\mathcal{S}}\, x$.
(Comp Object) $[\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]\ \widehat{\mathcal{S}}\ [\ell_i = \varsigma(x_i)b_i'\ ^{i \in 1..n}]$ if $b_i\ \mathcal{S}\ b_i'$ for $i \in 1..n$.
(Comp Select) $a.\ell\ \widehat{\mathcal{S}}\ a'.\ell$ if $a\ \mathcal{S}\ a'$.
(Comp Update) $a.\ell \Leftarrow \varsigma(x)b\ \widehat{\mathcal{S}}\ a'.\ell \Leftarrow \varsigma(x)b'$ if $a\ \mathcal{S}\ a'$ and $b\ \mathcal{S}\ b'$.
(Comp Clone) $clone(a)\ \widehat{\mathcal{S}}\ clone(a')$ if $a\ \mathcal{S}\ a'$.
(Comp Let) $let\ x = a\ in\ b\ \widehat{\mathcal{S}}\ let\ x = a'\ in\ b'$ if $a\ \mathcal{S}\ a'$ and $b\ \mathcal{S}\ b'$.

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

Proposition 34
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 $a \approx a'$. 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.
(Comp Clone) Here $a = clone(a_0)$, $a' = clone(a_0')$, and $a_0 \approx a_0'$. But then $a \approx a'$ is immediate from Lemma 29 with $\mathcal{R} = clone(\bullet)$.
(Comp Select) Immediate from Lemma 29 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 Lemma 29, $a_0 \approx a'_0$ implies $a_0.\ell \Leftarrow \varsigma(x)b \approx a'_0.\ell \Leftarrow \varsigma(x)b$. Because $\approx$ is transitive the result follows if $a'_0.\ell \Leftarrow \varsigma(x)b \approx a'_0.\ell \Leftarrow \varsigma(x)b'$. By Lemma 30, this again follows if

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

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

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

where $\iota = \rho(y)$. If the object $\sigma(\iota)$ has no ℓ method, both configurations are stuck (do not converge) and the equivalence holds by Lemma 25. Otherwise it follows by Lemma 24 if

$$(\iota, \sigma_1) \sim_{dom(\sigma)} (\iota, \sigma'_1)$$

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, we must show that

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

for all $\mathscr{R}$ with $locs(\mathscr{R}) \subseteq dom(\sigma)$ and $fv(\mathscr{R}) = \{\bullet\}$. Let relation $\mathscr{T}$ 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, $\sigma_1\ \mathscr{T}\ \sigma'_1$. We shall argue that

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

Suppose $(a, \sigma)\downarrow$, that is, there exist n and a terminal configuration d such that $(a, \sigma) \to^n d$. 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 then (a, σ') is terminal too.
Otherwise there exists (a_1, σ_1) such that $(a, \sigma) \to (a_1, \sigma_1) \to^{n-1} d$. By inspection of the reduction rules we see that $(a, \sigma') \to (a_1, \sigma'_1)$ with $\sigma_1\ \mathscr{T}\ \sigma'_1$, unless a is of the form $a = \mathscr{R}[\iota.\ell]$ where $\sigma(\iota)$ and $\sigma'(\iota)$ have methods $\ell = \varsigma(x)b\rho$ and $\ell = \varsigma(x)b'\rho$, respectively. In that case $(a_1, \sigma_1) = (\mathscr{R}[b\rho'], \sigma)$ and $(a, \sigma') \to (\mathscr{R}[b'\rho'], \sigma')$ where $\rho' = (x \mapsto \iota) :: \rho$. Since $(\mathscr{R}[b\rho'], \sigma) \to^{n-1} d$ in one less step than $(a, \sigma) \to^n d$, we get $(\mathscr{R}[b\rho'], \sigma')\downarrow$ by the induction hypothesis. Moreover, $b \approx b'$ implies $(b\rho', \sigma') \sim_{dom(\sigma')} (b'\rho', \sigma')$. Hence $(\mathscr{R}[b\rho'], \sigma') \updownarrow (\mathscr{R}[b'\rho'], \sigma')$ and we obtain $(\mathscr{R}[b'\rho'], \sigma')\downarrow$ and $(a, \sigma')\downarrow$, as required.
This completes the induction on n and we conclude that $(a, \sigma)\downarrow$ implies $(a, \sigma')\downarrow$. The reverse implication is symmetrical. So $(a, \sigma) \updownarrow (a, \sigma')$, as required.

(Comp Object) Follows from case (Comp Update) by repeated applications of Proposition 32(3).

(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 Lemma 29. 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 31(1) and Lemma 30. Finally, $a \approx a'$ because $\approx$ is transitive. $\square$

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 35

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

Proof

It is easy to see from the definition that operational equivalence is adequate. Conversely, whenever $\mathcal{S}$ is compatible and adequate and $a\ \mathcal{S}\ a'$, one can show that $(\mathcal{R}[a]\rho, \sigma) \updownarrow (\mathcal{R}[a']\rho, \sigma)$ by constructing static terms b and b' from a and a' such that $b\ \mathcal{S}\ b'$, by compatibility, and $(b, []) \to^* (\mathcal{R}[a]\rho, \sigma)$ and $(b', []) \to^* (\mathcal{R}[a']\rho, \sigma)$. The desired conclusion then follows by determinacy of reduction and adequacy. $\square$

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

Operational (CIU) equivalence coincides with contextual equivalence.

Proof

We must prove that $a \approx a'$ if and only if $\{(a, a')\}^c$ is adequate. 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. $\square$

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 1, 2 and 3 imply that adequacy can equivalently be defined on the 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 (see the discussion below).

4.6 Discussion and related work

The store model

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, it was easy to prove Proposition 33(4):

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

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 example, without this constraint, there is a closing instantiation of the above equation such that one side converges while the other diverges. Take $b = x$, $b' = x'.\ell'$, and $a = z.\ell'$, and substitute the object $[\ell \mapsto \iota]$ for y, and the object $[\ell' \mapsto \iota]$ for z, two objects that share the method ι but that are not identical. Now, if we run each side in the method store $[\iota \mapsto \varsigma(x)[]]$, we find that the left hand side 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,

$$\mathit{let}\ x = []\ \mathit{in}\ [\ell = \varsigma(s)x] \quad \approx \quad [\ell = \varsigma(s)[]] \tag{3}$$

is an instance of Proposition 31(1) because [] is a value. In our object store model, the proof of (3) becomes somewhat involved and requires a tedious argument analogous to that of Lemma 27.

Functions

To keep the exposition simple and focused on imperative objects, the theory of operational equivalence is only presented for the core calculus. The definition of operational equivalence and the results for the core calculus can be extended to the full calculus with functions considered in the previous sections, along the lines of the similar work on a λ-calculus with references by Honsell, Mason, Smith and Talcott (1993). All the laws in section 4.3 remain valid for the full calculus. Nonetheless, the extension of the theory of operational equivalence is not conservative; for instance, $(let\ y = clone(z)\ in\ [\,]) \approx [\,]$ is a valid equation in the theory for the core calculus, where every value is an object location, but not in the theory for the full calculus, where z may be instantiated to a function value $\lambda(x)b$ and $(let\ y = clone(\lambda(x)b)\ in\ [\,], \sigma)$ is stuck whereas $([\,], \sigma)$ terminates for any store σ.

Related work

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 (1998) for an example of this for the stateless object calculus of Abadi and Cardelli (1996). Talcott (1998) 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.

Some transformations for rearranging side effects are rather cumbersome to express in terms of equational laws as they depend on variables being bound to distinct locations. We have not pursued this issue in great depth. For further study it would be interesting to consider program logics such as VTLoE (Honsell *et al.*, 1993) or specification logic (Reynolds, 1982; Reddy, 1998), where it is possible to express such conditions directly.

Earlier work on operational equivalence for object calculi has been concerned with stateless objects. For instance, Gordon and Rees (1996) and Gordon (1998) characterise contextual equivalence exactly via forms of bisimilarity induced by the primitive operational semantics of objects. See Stark (1997) for an account of the difficulties of defining bisimulation in the presence of imperative effects.

In recent work, Kleist and Sangiorgi (1998) translate the first-order typed imperative object calculus into a typed π-calculus. Among other results, they verify typed versions of some of our laws by translation into bisimilar π-calculus processes. In comparison, working directly with the operational semantics as we do seems to be simpler than establishing and reasoning about an encoding.

The main influence on this section has been the literature on operational theories for functional languages with state. Our experience is that existing techniques for functional languages with state scale up well to deal with the object-oriented features of the imperative object calculus. CIU equivalence was introduced by Mason and Talcott (1991) and has been the topic of much research; see Talcott (1998) 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; 1995). However, 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. Many subtleties of the theory of operational equivalence are shared with theories for functional languages with state, including the examples of Meyer and Sieber (1988). These subtleties have been addressed by advanced operational methods (Honsell *et al.*, 1993; Pitts and Stark, 1998) which 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 *et al.*, 1997; Jones, 1996; Walker, 1995; Sangiorgi, 1997), but the technique of CIU equivalence was not used in these studies.

5 Refinement: 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 its correctness, Theorem 5.

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, give an example in section 5.2.1, and prove the correctness of the algorithm 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.

$$a, b ::= \dots \mid a.j \mid a.j \Leftarrow \varsigma(x)b \qquad\qquad \text{terms, } 0 < j$$

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

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:

$\mathscr{R} ::= \dots \mid \mathscr{R}.j \mid \mathscr{R}.j \Leftarrow \varsigma(x)b$	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 axioms and rules:

(Red Offset Select) $(\mathscr{R}[\iota.j], \sigma) \rightarrow (\mathscr{R}[b_j\{\!\{\iota/x_j\}\!\}], \sigma)$
if $\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i\ {}^{i \in 1..n}]$ and $j \in 1..n$.
(Red Offset Update) $(\mathscr{R}[\iota.j \Leftarrow \varsigma(x)b], \sigma) \rightarrow (\mathscr{R}[\iota], \sigma')$
if $\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i\ {}^{i \in 1..n}]$, $j \in 1..n$ and
$\sigma' = \sigma + (\iota \mapsto [\ell_i = \varsigma(x_i)b_i\ {}^{i \in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i\ {}^{i \in j+1..n}])$.

(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)

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

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.

$A, B ::= [\ell_i\ {}^{i \in 1..n}]$	layout type, ℓ_i distinct
$E ::= [x_i \mapsto A_i\ {}^{i \in 1..n}]$	environment, x_i distinct

For an object $o = [\ell_i = \varsigma(x_i)b_i\ {}^{i \in 1..n}]$, let $\mathit{layout}(o) = [\ell_i\ {}^{i \in 1..n}]$.

The algorithm infers 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 $resolve(E, a)$, which takes an environment E and a static term a with $fv(a) \subseteq dom(E)$, and produces a pair (a', A), where static term a' is the residue of a after resolution of labels known from layout types 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.

$$\begin{array}{l}
resolve(E, x) \overset{\text{def}}{=} (x, E(x)) \quad \text{where } x \in dom(E) \\
resolve(E, [\ell_i = \varsigma(x_i)a_i\, {}^{i \in 1..n}]) \overset{\text{def}}{=} ([\ell_i = \varsigma(x_i)a'_i\, {}^{i \in 1..n}], A) \\
\quad \text{where } A = [\ell_i\, {}^{i \in 1..n}] \\
\quad \text{and } (a'_i, B_i) = resolve((x_i \mapsto A) :: E, a_i),\ x_i \notin dom(E), \text{ for each } i \in 1..n \\
resolve(E, a.p) \overset{\text{def}}{=} \\
\quad \begin{cases} (a'.j, []) & \text{if } j \in 1..n \text{ and } p = \ell_j \\ (a'.p, []) & \text{otherwise} \end{cases} \\
\qquad \text{where } (a', [\ell_i\, {}^{i \in 1..n}]) = resolve(E, a) \\
resolve(E, a.p \Leftarrow \varsigma(x)b) \overset{\text{def}}{=} \\
\quad \begin{cases} (a'.j \Leftarrow \varsigma(x)b', A) & \text{if } j \in 1..n \text{ and } p = \ell_j \\ (a'.p \Leftarrow \varsigma(x)b', A) & \text{otherwise} \end{cases} \\
\qquad \text{where } (a', A) = resolve(E, a),\ A = [\ell_i\, {}^{i \in 1..n}] \\
\qquad \text{and } (b', B) = resolve((x \mapsto A) :: E, b),\ x \notin dom(E) \\
resolve(E, clone(a)) \overset{\text{def}}{=} (clone(a'), A) \quad \text{where } (a', A) = resolve(E, a) \\
resolve(E, let\ x = a\ in\ b) \overset{\text{def}}{=} (let\ x = a'\ in\ b', B) \\
\quad \text{where } (a', A) = resolve(E, a) \\
\quad \text{and } (b', B) = resolve((x \mapsto A) :: E, b),\ x \notin dom(E)
\end{array}$$

5.2.1 *Example of static resolution*

To illustrate the algorithm in action, consider the object $pair(x, y)$:

$$\begin{array}{r}
[fst = \varsigma(s)x, snd = \varsigma(s)y, swap = \varsigma(s)let\ x = s.fst\ in\ let\ y = s.snd\ in \\
(s.fst \Leftarrow \varsigma(s')y).snd \Leftarrow \varsigma(s')x]
\end{array}$$

Then, for arbitrary layout types A and B,

$$resolve([x \mapsto A, y \mapsto B], pair(x, y)) = (pair'(x, y), [fst, snd, swap])$$

where $pair'(x, y)$ denotes the object:

$$\begin{array}{r}
[fst = \varsigma(s)x, snd = \varsigma(s)y, swap = \varsigma(s)let\ x = s.1\ in\ let\ y = s.2\ in \\
(s.1 \Leftarrow \varsigma(s')y).2 \Leftarrow \varsigma(s')x]
\end{array}$$

All method selects and method updates in the object have been statically resolved. The layout type $[fst, snd, swap]$ asserts that $pair(x, y)$ and $pair'(x, y)$ will evaluate to objects with this layout. This means, not surprisingly, that any select or update of fst, snd or $swap$ on $pair(x, y)$ are statically resolved. For instance:

$$resolve([x \mapsto A, y \mapsto B], pair(x, y).swap) = (pair'(x, y).3, [])$$

Here, the empty layout type $[]$ asserts that nothing is known about the layout of the objects returned by $pair(x,y).swap$ and $pair'(x,y).3$. So, if we select $swap$ twice, the second method select is not resolved:

$$resolve([x \mapsto A, y \mapsto B], pair(x,y).swap.swap) = (pair'(x,y).3.swap, [])$$

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 a'.

(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 $x_i \notin dom(E)$)

$$\frac{((x_i \mapsto B) :: E, \Sigma) \vdash a_i \leftrightarrow a_i' : A_i \quad \forall i \in 1..n}{(E,\Sigma) \vdash [\ell_i = \varsigma(x_i)a_i \;{}^{i \in 1..n}] \leftrightarrow [\ell_i = \varsigma(x_i)a_i' \;{}^{i \in 1..n}] : B}$$

(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 $j \in 1..n$)

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

(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 $x \notin dom(E)$, $A = [\ell_i{}^{i \in 1..n}]$ and $j \in 1..n$)

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

(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 38, 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 36
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). $\square$

For illustration, let us revisit the pair example from section 5.1. Via (Layout Object), (Layout x), (Layout Let), (Layout Select 3) and (Layout Update 3) we may derive:

$$([x \mapsto A, y \mapsto B],[]) \vdash pair(x,y) \leftrightarrow pair'(x,y) : [fst, snd, swap]$$

Further, via (Layout Select 3) and (Layout Select 1) we derive:

$$([x \mapsto A, y \mapsto B],[]) \vdash pair(x,y).swap \leftrightarrow pair'(x,y).3 : []$$

$$([x \mapsto A, y \mapsto B],[]) \vdash pair(x,y).swap.swap \leftrightarrow pair'(x,y).3.swap : []$$

We will make precise the connection between evaluation and resolution in Proposition 38. 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 $dom(\Sigma) = dom(\sigma) = dom(\sigma')$)

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

(Layout Config)

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

For example, consider the store $\sigma = [\iota_1 \mapsto o_1, \iota_2 \mapsto o_2]$ and a store type $\Sigma = [\iota_1 \mapsto A_1, \iota_2 \mapsto A_2]$ such that $\vdash \sigma \leftrightarrow \sigma : \Sigma$. Then, using the rules above, we may derive:

$$\vdash (pair(\iota_1, \iota_2).swap, \sigma) \leftrightarrow (pair'(\iota_1, \iota_2).3, \sigma) : ([], \Sigma)$$

where $pair'(\iota_1, \iota_2)$ is the object $pair(\iota_1, \iota_2)$ with all labels resolved, as in the previous example. Given the set of rules defining the resolution relation, we cannot derive a layout type other than $[]$ for $pair(\iota_1, \iota_2).swap$ and $pair'(\iota_1, \iota_2).3$.

To see the effect of evaluation on the layout type of these configurations, we derive:

$$(pair(x, y).swap, \sigma) \Downarrow (\iota, (\iota \mapsto pair(\iota_2, \iota_1)) :: \sigma)$$

and

$$(pair'(x, y).3, \sigma) \Downarrow (\iota, (\iota \mapsto pair'(\iota_2, \iota_1)) :: \sigma)$$

where $\iota \notin dom(\sigma)$, by the evaluation rules from sections 2.3 and 5.1. Moreover, using the rules above, we may derive:

$$\vdash (\iota, (\iota \mapsto pair(\iota_2, \iota_1)) :: \sigma) \leftrightarrow (\iota, (\iota \mapsto pair'(\iota_2, \iota_1)) :: \sigma) : (A, (\iota \mapsto A) :: \Sigma)$$

where $A = [fst, snd, swap]$.

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 36, but also because we can prove that resolution is preserved by evaluation, Proposition 38. We first need the following substitution lemma.

Lemma 37
$(E'@[x \mapsto A]@E'', \Sigma) \vdash a \leftrightarrow a' : B$, $\iota \in dom(\Sigma)$ 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 $\Sigma \leqslant \Sigma'$ if and only if $dom(\Sigma) \subseteq dom(\Sigma')$ and $\Sigma(\iota) = \Sigma'(\iota)$ for each $\iota \in dom(\Sigma)$.

Proposition 38
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 $\Sigma \leqslant \Sigma'$. 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 $\Sigma \leqslant \Sigma'$. Moreover, $A \neq []$ implies $A = A'$.

Proof
Part (1) is by induction on the derivation of $c \Downarrow d$. Part (2) is by induction on the derivation of $c' \Downarrow d'$. □

Lemma 39
Suppose $([x_i \mapsto []\ ^{i \in 1..n}], []) \vdash a \leftrightarrow a' : A$. Consider any reduction context $\mathscr{R}$ with $locs(\mathscr{R}) = \varnothing$ such that $fv(\mathscr{R}) - \{\bullet, x_1, \ldots, x_n\} = \{x_{n+1}, \ldots, x_{n+m}\}$. Then $([x_i \mapsto []\ ^{i \in 1..n+m}], []) \vdash \mathscr{R}[a] \leftrightarrow \mathscr{R}[a'] : B$ for some B.

Proof
By induction on the size of the reduction context $\mathscr{R}$. □

Lemma 40
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 $([x_i \mapsto []\ ^{i \in 1..n}], []) \vdash a \leftrightarrow a' : B$. □

Theorem 5
Suppose a is a static term with free variables $x_1, \ldots, x_n$. If routine *resolve* $([x_i \mapsto []\ ^{i \in 1..n}], a)$ returns (a', A), then $a \approx a'$.

Proof
By Lemma 28, to show $a \approx a'$, it suffices to prove $(\mathscr{R}[a]\rho, \sigma) \updownarrow (\mathscr{R}[a']\rho, \sigma)$, for all static reduction contexts $\mathscr{R}$, well formed stores σ, and substitutions $\rho : fv(\mathscr{R}[a]) \cup fv(\mathscr{R}[a']) \to dom(\sigma)$. Consider any static reduction context $\mathscr{R}$, any well formed store σ and any substitution $\rho : fv(\mathscr{R}[a]) \cup fv(\mathscr{R}[a']) \to dom(\sigma)$. Let $E = [x_i \mapsto []\ ^{i \in 1..n}]$ and $E' = [x_i \mapsto []\ ^{i \in 1..n+m}]$ where $\{x_{n+1}, \ldots, x_{n+m}\} = fv(\mathscr{R}) - \{\bullet, x_1, \ldots, x_n\}$. By Lemma 36, we may derive $(E, []) \vdash a \leftrightarrow a' : A$. By Lemma 39, $(E, []) \vdash a \leftrightarrow a' : A$ implies $(E', []) \vdash \mathscr{R}[a] \leftrightarrow \mathscr{R}[a'] : B$ for some B. If $\sigma = [\iota_i = o_i\ ^{i \in 1..n}]$, let $\Sigma = [\iota_i = layout(o_i)\ ^{i \in 1..n}]$. By Lemma 40, $(E', []) \vdash \mathscr{R}[a] \leftrightarrow \mathscr{R}[a'] : B$ and $\rho : \{x_1, \ldots, x_{n+m}\} \to dom(\Sigma)$ imply $([], \Sigma) \vdash \mathscr{R}[a]\rho \leftrightarrow \mathscr{R}[a']\rho : B'$ for some B'. By (Layout Store), $\Sigma \vdash \sigma \leftrightarrow \sigma$. Hence by (Layout Config), we have $\vdash (\mathscr{R}[a]\rho, \sigma) \leftrightarrow (\mathscr{R}[a']\rho, \sigma)$. Suppose that $(\mathscr{R}[a]\rho, \sigma)\downarrow$. By Theorem 1 there is c with $(\mathscr{R}[a]\rho, \sigma) \Downarrow c$. By Proposition 38(1), $\vdash (\mathscr{R}[a]\rho, \sigma) \leftrightarrow (\mathscr{R}[a']\rho, \sigma)$ implies there is c' such that $(\mathscr{R}[a']\rho, \sigma) \Downarrow c'$, and therefore $(\mathscr{R}[a']\rho, \sigma)\downarrow$, again by Theorem 1. Similarly, by Proposition 38(2) and $\vdash (\mathscr{R}[a]\rho, \sigma) \leftrightarrow (\mathscr{R}[a']\rho, \sigma)$, $(\mathscr{R}[a']\rho, \sigma)\downarrow$ implies $(\mathscr{R}[a]\rho, \sigma)\downarrow$. Therefore $(\mathscr{R}[a]\rho, \sigma) \updownarrow (\mathscr{R}[a']\rho, \sigma)$, 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.

Two alternative approaches to program analysis for untyped object calculi are a control flow analysis for the imperative object calculus, expressed as a flow logic (Nielson and Nielson, 1998) and a set-based control flow analysis for a concurrent, imperative object calculus (di Blasio *et al.*, 1997). Both should be adaptable to the problem of statically resolving method offsets. These approaches are rather more complex than ours but may lead to more precise results.

6 Conclusions

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

First, we presented both a big-step and a small-step substitution-based operational semantics for the calculus and proved them equivalent to a closure-based operational semantics like that given by Abadi and Cardelli (Theorem 1 and Theorem 2).

Next, we designed an object-oriented abstract machine as a straightforward extension of Leroy's abstract machine with instructions for manipulating objects. Our third result is a correctness proof for the abstract machine and its compiler (Theorem 3). 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 fourth result is that Mason and Talcott's CIU equivalence coincides with Morris-style contextual equivalence (Theorem 4). This is the first result about program equivalence for the imperative object calculus, a topic left unexplored by Abadi and Cardelli's book. The selection of laws of program equivalence that we establish is a first step towards an algebra of imperative objects that may be useful for future work on imperative object-oriented languages. Already, typed versions of some of our laws have been verified for a typed imperative object calculus (Kleist and Sangiorgi, 1998).

One benefit of CIU equivalence is that it allows the verification of compiler optimisations. We illustrate this by proving that an optimisation algorithm from our implementation preserves contextual equivalence (Theorem 5).

Acknowledgements

Martín Abadi, Greg Sullivan, Carolyn Talcott and several anonymous referees commented on previous versions of this paper. Arthur Nunes pointed out an error in the unloading machine in section 3. During the course of this work, Gordon held a Royal Society University Research Fellowship and Hankin held an EPSRC Research Studentship. Lassen was supported by a grant from the Danish Natural Science Research Council.

A Glossary of notation

Notation from Section 2

a, b	term
u, v	value
s	substitution (of values for variables)
o	object
σ	store
c, d	configuration
ι	location in store
ℓ	method label
$\mathscr{R}$	reduction context
$\bullet$	hole
ϕ	phrase of syntax
U, V	closure-based value
S	closure-based stack
O	closure-based object
Σ	closure-based store

C, D	closure-based configuration
$fv(\phi)$	variables free in ϕ
$\phi\{\!\{\psi/x\}\!\}$	substitution
$\phi_i\,{}^{i\in 1..n}$	$\phi_1, \ldots, \phi_n$
$locs(\phi)$	locations occurring in ϕ
$[\phi_i\,{}^{i\in 1..n}]$	list
$\phi :: [\phi_i\,{}^{i\in 1..n}]$	list constructor
$[\phi_i\,{}^{i\in 1..n}]$ @ $[\phi'_i\,{}^{i\in 1..m}]$	list concatenation
$[x_i \mapsto \phi_i\,{}^{i\in 1..n}]$	finite map
$dom(f)$	domain of map
$f + (x \mapsto \phi)$	extension of map
$c \to d$	small-step reduction
$c \searrow d$	reduction to terminal configuration
$\vdash \sigma\ ok$	store well formed
$\vdash c\ ok$	configuration well formed
$\vdash_w c\ ok$	configuration well formed at w
$c \equiv_w c'$	structural equivalence at w
$\mathscr{R}[a]$	substitution of a for hole in $\mathscr{R}$
$c \Downarrow d$	big-step evaluation
$C \Downarrow D$	closure-based big-step evaluation
$V \leadsto v$	value unloading
$S \leadsto s$	stack unloading
$O \leadsto o$	object unloading
$\Sigma \leadsto \sigma$	store unloading
$C \leadsto c$	configuration unloading

Notation from Section 3

op	machine operation
ops	operation list
P, Q	machine state
U, V	value
$U^\diamond, V^\diamond$	value or mark
E	environment
AS	argument stack
RS	return stack
F	closure
O	stored object
Σ	store
p, q	unloading machine state
e	unloading environment
as	unloading argument stack
$a^\diamond, b^\diamond$	term or mark
$u^\diamond, v^\diamond$	value or mark
$xs \vdash a \Rightarrow ops$	compilation judgment

$\mathtt{grab}^n$	list of `grab` instructions
$((ops, E, AS, RS), \Sigma)$	configuration
(ops, e, as, RS)	unloading machine state
$C \xrightarrow{\beta} D$	β-transition
$C \xrightarrow{\tau} D$	τ-transition
$C \xrightarrow{\beta\tau} D$	either β or τ transition
$C \searrow D$	reduction to terminal configuration
$\diamond$	mark on stack
$\mathit{fun}(ops, E)$	function closure
$p \xrightarrow{u} q$	unloading machine step
$(ops, e) \leadsto (x)a$	unload abstraction
$\mathit{fun}(ops, e) \leadsto \lambda(x)b$	unload function
$[U_i^{\diamond\ i \in 1..n}] \leadsto [u_i^{\diamond\ i \in 1..n}]$	unload list
$O \leadsto o$	unload object
$\Sigma \leadsto \sigma$	unload store
$C \leadsto c$	unload configuration
$p \leadsto a$	unload unloading machine state
$E \leadsto e$	unload environment
$AS \leadsto as$	unload argument stack
$P \downarrow p$	conversion of machine state to unloading machine state
shape p	shape of unloading machine state

Notation from Section 4

ρ	substitution (of locations for variables)
$\mathcal{S}$	relation on terms
$c \updownarrow c'$	same-convergence relation
$c \sim_w c'$	experimental equivalence at w
$a \approx a'$	operational equivalence
$\widehat{\mathcal{S}}$	compatible refinement
$\mathcal{S}^c$	context closure

Notation from Section 5

A, B	layout type
E	environment
Σ	store type
$\mathit{layout}(o)$	layout of object
$\mathit{resolve}(E, a)$	static resolution algorithm
$(E, \Sigma) \vdash a \leftrightarrow a' : A$	resolution relation on terms
$\vdash \sigma \leftrightarrow \sigma' : \Sigma$	resolution relation on stores
$\vdash c \leftrightarrow c' : (A, \Sigma)$	resolution relation on configurations
$\Sigma \leqslant \Sigma'$	store comparison

References

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

Abadi, M. and Cardelli, L. (1995*b*) An imperative object calculus: Basic typing and soundness. *Proceedings SIPL'95*. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign.

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

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

Agha, G., Mason, I., Smith, S. and Talcott, C. (1997) A foundation for actor computation. *J. Functional Programming*, **7**(1), 1–72.

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

Chambers, C. (1992) *The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages*. PhD thesis, Computer Science Department, Stanford University.

di Blasio, P., Fisher, K. and Talcott, C. (1997) Analysis for concurrent objects. In: Bowman, H. and Derrick, J. (eds.), *Proceedings FMOODS'97*. Chapman & Hall.

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

Felleisen, M. and Friedman, D. (1989) A syntactic theory of sequential state. *Theoretical Comput. Sci.* **69**, 243–287.

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

Gordon, A. D. (1998) Operational equivalences for untyped and polymorphic object calculi. In: Gordon, A. D. and Pitts, A. M. (eds.), *Higher Order Operational Techniques in Semantics*. Cambridge University Press.

Gordon, A. D. and Hankin, P. D. (1998) A concurrent object calculus: Reduction and typing. *Proceedings HLCL'98*. Elsevier.

Gordon, A. D. and Pitts, A. M. (eds). (1998) *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge University Press.

Gordon, A. D. and Rees, G. D. (1996) Bisimilarity for a first-order calculus of objects with subtyping. *Proceedings POPL'96*, pp. 386–395. ACM.

Gordon, A. D., Lassen, S. B. and Hankin, P. D. (1998) Compilation and equivalence of imperative objects (revised report). BRICS Report RS–98–55, BRICS, Department of Computer Science, University of Aarhus. A shorter version was presented at *Foundations of Software Technology and Theoretical Computer Science, 17th Conference*, Kharagpur, India, December 1997. *Lecture Notes of Computer Science 1346*, pp. 74–87. Springer-Verlag.

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

Hannan, J. and Miller, D. (1992) From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, **4**(2), 415–489.

Hardin, T., Maranget, L. and Pagano, B. (1998) Functional runtime systems within the lambda-sigma calculus. *J. Functional Programming*, **8**(2), 131–176.

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

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

Jones, C. B. (1996) Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, **8**(2), 105–122.

Kahn, G. (1987) Natural semantics. *Proceedings STACS'87: Lecture Notes in Computer Science 247*, pp. 22–39. Springer-Verlag.

Kleist, J. and Sangiorgi, D. (1998) Imperative objects and mobile processes. *Proceedings PROCOMET'98.*

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

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

Martin-Löf, P. (1983) Notes on the domain interpretation of type theory. *Proceedings of the Workshop on Semantics of Programming Languages*, Chalmers, Sweden.

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

Mason, I. and Talcott, C. (1992) References, local variables and operational reasoning. *Proceedings LICS'92.*

Mason, I. and Talcott, C. (1995) Reasoning about object systems in VTLoE. *Int. J. Foundations of Comput. Sci.* **6**(3), 265–298.

Meyer, A. and Cosmadakis, S. (1988) Semantical paradigms: Notes for an invited lecture. *Proceedings LICS'88*, pp. 236–253.

Meyer, A. and Sieber, K. (1988) Towards fully abstract semantics for local variables. *Proceedings POPL'88*, pp. 236–253.

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

Moggi, E. (1989) Notions of computations and monads. *Information & Computation*, **93**, 55–92.

Morris, J. H. (1968) *Lambda-calculus models of programming languages.* PhD thesis, MIT.

Nielson, F. and Nielson, H. R. (1998) The flow logic of imperative objects. *Proceedings MFCS'98: Lecture Notes in Computer Science.* Springer-Verlag.

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

Ohori, A. (1992) A compilation method for ML-style polymorphic record calculi. *Proceedings POPL'92*, pp. 154–165. ACM.

Pitts, A. M. and Stark, I. D. B. (1998) Operational reasoning for functions with local state. In: Gordon, A. D. and Pitts, A. M. (eds.), *Higher Order Operational Techniques in Semantics.* Cambridge University Press.

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

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

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

Reddy, U. S. (1998) Objects and classes in Algol-like languages. *Proceedings FOOL'98.*

Reynolds, J. C. (1982) Idealized Algol and its specification logic. In: Néel, D. (ed.), *Tools and Notions for Program Construction*, pp. 121–161. Cambridge University Press.

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

Sangiorgi, D. (1997) Typed π-calculus at work: a proof of Jones' parallelisation transformation on concurrent objects. *Proceedings FOOL'97.*

Sestoft, P. (1997) Deriving a lazy abstract machine. *J. Functional Programming*, **3**(7), 231–264.

Stark, I. (1997) Names, equations, relations: Practical ways to reason about *new*. *Proceedings TLCA'97: Lecture Notes in Computer Science 1210*, pp. 336–353. Springer-Verlag.

Talcott, C. (1998) Reasoning about functions with effects. In: Gordon, A. D. and Pitts, A. M. (eds.), *Higher Order Operational Techniques in Semantics*. Cambridge University Press.

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

Wand, M. (1995) Compiler correctness for parallel languages. *Proceedings FPCA'95*, pp. 120–134. ACM.