Compilation and Equivalence of Imperative Objects

A.D. Gordon[1], P.D. Hankin[1], and S.B. Lassen[2]

[1] Computer Laboratory, University of Cambridge
[2] BRICS, Computer Science Department, University of Aarhus

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. Our main result is a direct proof, via a small-step unloading machine, of the correctness of compilation to a closure-based abstract machine. Our second result is that contextual equivalence of objects coincides with a form of Mason and Talcott's CIU equivalence; the latter provides a tractable means of establishing operational equivalences. Finally, we prove correct an algorithm, used in our prototype compiler, for statically resolving method offsets. This is the first study of correctness of an object-oriented abstract machine, and of CIU equivalence for an object-oriented language.

1 Motivation

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. The language we describe is essentially the untyped imperative object calculus of Abadi and Cardelli [1–3], 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 concurrent object-oriented languages. This paper develops formal foundations and verification methods to document and better understand various aspects of our implementation. Our work recasts techniques originating in studies of the λ-calculus in the setting of the imperative object calculus. In particular, our reduction relation for the object calculus, our design of an object-oriented abstract machine, our compiler correctness proof and our notion of program equivalence are all based on earlier studies of the λ-calculus. This paper is the first application of these techniques to an object calculus and shows they may easily be re-used in an object-oriented setting.

Our system compiles the imperative object calculus to bytecodes for an abstract machine, implemented in C, based on the ZAM of Leroy's CAML Light

[16]. 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.

In Section 2 we present the imperative object calculus together with a small-step substitution-based operational semantics. Section 3 gives a formal description of an object-oriented abstract machine, a simplification of the machine used in our implementation. We present a compiler from the object calculus to instructions for the abstract machine. We prove the compiler correct by adapting a proof of Rittri [23] to cope with state and objects. In Section 4, we develop a theory of operational equivalence for the imperative object calculus, based on the CIU equivalence of Mason and Talcott [18]. We establish useful equivalence laws and prove that CIU equivalence coincides with Morris-style contextual equivalence [20]. In Section 5, we exercise operational equivalence by specifying and verifying a simple optimisation that resolves at compile-time certain method labels to integer offsets. We discuss related work at the ends of Sections 3, 4 and 5. Finally, we review the contributions of the paper in Section 6.

The full version of this paper, with proofs, is available as a technical report [9].

2 An Imperative Object Calculus

We begin with the syntax of an untyped imperative object calculus, the **imp**$\boldsymbol{\varsigma}$ calculus of Abadi and Cardelli [3] augmented to include store locations as terms. Let x, y, and z range over an infinite collection of *variables*. Let ι range over an infinite collection of *locations*, the addresses of objects in the store.

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

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

Informally, when an object is created, it is put at a fresh location, ι, in the store, and referenced thereafter by ι. Method selection runs the body of the method with the self parameter (the x in $\varsigma(x)b$) bound to the location of the object containing the method. Method update allows 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 the book of Abadi and Cardelli [3] 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 since the order of methods in an object is important for an algorithm we present in Section 5 for statically resolving method offsets. 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 an example of programming in the imperative object calculus, 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.

Before proceeding with the formal semantics for the calculus, we fix notation for finite lists and finite maps. We write finite lists in the form $[\phi_1, \ldots, \phi_n]$, which we usually write as $[\phi_i \ {}^{i \in 1..n}]$. Let $\psi :: [\phi_i \ {}^{i \in 1..n}] = [\psi, \phi_i \ {}^{i \in 1..n}]$. Let $[\phi_i \ {}^{i \in 1..m}]@[\psi_j \ {}^{j \in 1..n}] = [\phi_i \ {}^{i \in 1..m}, \psi_j \ {}^{j \in 1..n}]$.

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

Now we specify a small-step substitution-based operational semantics for the calculus [8,18]. Let a *store*, σ, be a finite map $[\iota_i \mapsto o_i \ {}^{i \in 1..n}]$ 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. Let a *configuration*, c or d, be a pair (a, σ) where a is a term and σ is a store. Let a *reduction context*, $\mathcal{R}$, be a term given by the following grammar, with one free occurrence of a distinguished variable, $\bullet$:

$$\mathcal{R} ::= \bullet \mid \mathcal{R}.\ell \mid \mathcal{R}.\ell \Leftarrow \varsigma(x)b \mid clone(\mathcal{R}) \mid let\ x = \mathcal{R}\ in\ b$$

We write $\mathcal{R}[a]$ for the outcome of filling the single occurrence of the hole $\bullet$ in a reduction context $\mathcal{R}$ with the term a. Let the small-step substitution-based *reduction* relation, $c \to d$, be the smallest relation satisfying the following, where in each rule the hole in the reduction context $\mathcal{R}$ represents 'the point of execution'.

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

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

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

(Red Clone) $(\mathcal{R}[clone(\iota)], \sigma) \to (\mathcal{R}[\iota'], \sigma')$
if $\sigma(\iota) = o$, $\sigma' = (\iota' \mapsto o) :: \sigma$ and $\iota' \notin dom(\sigma)$.

(Red Let) $(\mathcal{R}[let\ x = \iota\ in\ b], \sigma) \to (\mathcal{R}[b\{\!\{\iota/x\}\!\}], \sigma)$.

Let a store σ be *well formed* if and only if $fv(\sigma(\iota)) = \varnothing$ and $locs(\sigma(\iota)) \subseteq dom(\sigma)$ for each $\iota \in dom(\sigma)$. Let a configuration (a, σ) be *well formed* if and only if $fv(a) = \varnothing$, $locs(a) \subseteq dom(\sigma)$ and σ is well formed. A routine case analysis shows that reduction sends a well formed configuration to a well formed configuration, and that reduction is deterministic up to the choice of freshly allocated locations in rules for object formation and cloning.

Let a configuration c be *terminal* if and only if there is a store σ and a location ι such that $c = (\iota, \sigma)$. We say a configuration c *converges* to d, $c \Downarrow d$, if and only if d is a terminal configuration and $c \to^* d$. Because reduction is deterministic, whenever $c \Downarrow d$ and c is well formed, the configuration d is unique up to the renaming of any newly generated locations in the store component of d.

Abadi and Cardelli define a big-step closure-based operational semantics for the calculus: it relates a configuration directly to the final outcome of taking many individual steps of computation, and it uses closures, rather than a substitution primitive, to link variables to their values. We find the small-step substitution-based semantics better suited for the proofs in Sections 3 and 5 as well as for developing the theory of operational equivalence in Section 4. We have proved, using an inductively defined relation unloading closures to terms, that our semantics is consistent with theirs in the following sense:

Proposition 1. *For any closed static term a, there is d such that $(a, [\,]) \Downarrow d$ if and only if evaluation of a converges in Abadi and Cardelli's semantics.*

3 Compilation to an Object-Oriented Abstract Machine

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

The instruction set of our abstract machine consists of the *operations*, ranged over by op, given as follows: $\mathtt{access}\ i$, $\mathtt{object}[(\ell_i, ops_i)^{\ i \in 1..n}]$ (ℓ_i distinct), $\mathtt{select}\ \ell$, $\mathtt{update}(\ell, ops)$ or $\mathtt{let}\ ops$, where ops ranges over operation lists.

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

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

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

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

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

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

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:

$P, Q ::= (ops, E, AC, RS)$	machine state
$E ::= [\iota_i^{\ i \in 1..n}]$	environment
$AC ::= [] \mid [\iota]$	accumulator
$RS ::= [F_i^{\ i \in 1..n}]$	return stack
$F ::= (ops, E)$	closure
$O ::= [(\ell_i, F_i)^{\ i \in 1..n}]$	stored object (ℓ_i distinct)
$\Sigma ::= [\iota_i \mapsto O_i^{\ i \in 1..n}]$	store (ι_i distinct)

In a configuration $((ops, E, AC, RS), \Sigma)$, ops is the current program. Environment E contains variable bindings. Accumulator AC either holds the result of evaluating a term, $AC = [\iota]$, or a dummy value, $AC = []$. Return stack RS holds return addresses during 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, either a method return or a variable lookup. Lemma 3 relates reductions of the object calculus and transitions of the abstract machine.

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

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

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

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

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

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

(β Let) $((\texttt{let}\; ops' :: ops, E, [\iota], RS), \Sigma) \xrightarrow{\beta} ((ops', \iota :: E, [], (ops, E) :: RS), \Sigma)$.

Each rule apart from the first tests whether the accumulator is empty or not. We can show that this test is always redundant when running code generated by our compiler. In the machine of the full version of this paper [9], we replace the accumulator with an argument stack, a list of values.

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 accumulator contains the term corresponding to the original machine state.

The unloading machine is like the abstract machine, except that instead of executing each instruction, it reconstructs the corresponding source term. Since no store lookups or updates are performed, the unloading machine does not act on a store. An unloading machine state is like an abstract machine state, except that locations are generalised to arbitrary terms. Let an *unloading machine state*, p or q, be a quadruple (ops, e, ac, RS) where e takes the form $[a_i \;^{i \in 1..n}]$ and ac takes the form $[]$ or $[a]$. Next we make a simultaneous inductive definition of a *u-transition* relation $p \xrightarrow{u} p'$ and an unloading relation, $(ops, e) \rightsquigarrow (x)b$, that unloads a closure to a method.

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

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

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

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

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

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

(u Return) $([], e, ac, (ops, E) :: RS) \xrightarrow{u} (ops, E, ac, RS)$.

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

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

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

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

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

We can prove the following:

Lemma 2. *Whenever* $[] \vdash a \Rightarrow ops$ *then* $((ops, [], [], []), []) \rightsquigarrow (a, [])$.

Lemma 3.
(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* d *such that* $D \rightsquigarrow d$ *and* $c \rightarrow d$

Let a big-step transition relation, $C \Downarrow D$, on machine states hold if and only if there are ι, E, Σ with $D = (([], E, [\iota], []), \Sigma)$ and $C\ (\xrightarrow{\beta} \cup \xrightarrow{\tau})^{*}\ D$.

Lemma 4.
(1) *If* $C \rightsquigarrow c$ *and* $C \Downarrow D$ *then there is* d *with* $D \rightsquigarrow d$ *and* $c \Downarrow d$
(2) *If* $C \rightsquigarrow c$ *and* $c \Downarrow d$ *then there is* D *with* $D \rightsquigarrow d$ *and* $C \Downarrow D$

Theorem 5. *Whenever* $[] \vdash a \Rightarrow ops$*, for all* d*,* $(a, []) \Downarrow d$ *if and only if there is* D *with* $((ops, [], [], []), []) \Downarrow D$ *and* $D \rightsquigarrow d$.

Proof. By Lemma 2 we have $((ops, [], [], []), []) \rightsquigarrow (a, [])$. Suppose $(a, []) \Downarrow d$. By Lemma 4(2), $((ops, [], [], []), []) \rightsquigarrow (a, [])$ and $(a, []) \Downarrow d$ imply there is D with $D \rightsquigarrow d$ and $((ops, [], [], []), []) \Downarrow D$. Conversely, suppose $((ops, [], [], []), []) \Downarrow D$ for some D. By Lemma 4(1), $((ops, [], [], []), []) \rightsquigarrow (a, [])$ and $((ops, [], [], []), []) \Downarrow D$ imply there is d with $D \rightsquigarrow d$ and $((ops, [], [], []), []) \Downarrow d$. $\square$

In the full version of this paper [9], we prove correct a richer machine, based on the machine used in our implementation, that supports functions as well as objects. The full machine has a larger instruction set than the one presented here, needs a more complex compiler and has an argument stack instead of an accumulator. The correctness proof is similar to the one for the machine presented here.

There is a large literature on proofs of interpreters based on abstract machines, such as Landin's SECD machine [12,22,25]. 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 [11], 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 [13] or process calculi [28]. Rather than introduce a metalanguage, we prove correctness of our abstract machine directly from its operational semantics. We adopted Rittri's idea [23] of unloading a machine state to a term via a specialised unloading machine. Our proof is simpler than Rittri's, and goes beyond it by dealing with state and objects.

Even in the full version of the paper there are differences, of course, between our formal model of the abstract machine and our actual implementation. One difference is that we have modelled programs as finitely branching trees, whereas in the implementation programs are tables of bytecodes indexed by a program counter. Another difference is that our model omits garbage collection, which is essential to the implementation. Therefore Theorem 5 only implies that the compilation strategy is correct; bugs may remain in its implementation.

4 Operational Equivalence of Imperative Objects

The standard operational definition of term equivalence is Morris-style contextual equivalence [20]: 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.

Mason and Talcott [18] prove that, for functional languages with state, contextual equivalence coincides with so-called CIU ("Closed Instances of Use") equivalence. Informally, two terms are CIU equivalent if and only if 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 work with in proofs.

In this section we adopt CIU equivalence as our notion of operational equivalence for imperative objects. We establish a variety of laws of equivalence. We show that operational equivalence is a congruence, and hence supports compositional equational reasoning. Finally, we prove that CIU equivalence coincides with contextual equivalence, as in Mason and Talcott's setting.

We define static terms a and a' to be *operationally equivalent*, $a \approx a'$, if, for all variables $x_1, \ldots, x_n$, all static reduction contexts $\mathcal{R}$ with $fv(\mathcal{R}[a]) \cup fv(\mathcal{R}[a']) \subseteq \{x_1, \ldots, x_n\}$, all well formed stores σ, and all locations $\iota_1, \ldots, \iota_n \in dom(\sigma)$, we have that configurations $(\mathcal{R}[a]\{\!\{\iota_i/x_i \; ^{i\in 1..n}\}\!\}, \sigma)$ and $(\mathcal{R}[a']\{\!\{\iota_i/x_i \; ^{i\in 1..n}\}\!\}, \sigma)$ either both converge or both do not converge.

It follows easily from the definition of operational equivalence that it is an equivalence relation on static terms and, moreover, that it is preserved by static reduction contexts:

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

From the definition of operational equivalence, it is possible to show a multitude of equational laws for the constructs of the calculus. For instance, the *let* construct satisfies laws corresponding to those of Moggi's computational λ-calculus [19], presented here in the form given by Talcott [27].

Proposition 6.
(1) $(let\ x = y\ in\ b) \approx b\{\!\{y/x\}\!\}$
(2) $(let\ x = a\ in\ \mathcal{R}[x]) \approx \mathcal{R}[a]$, *if* $x \notin fv(\mathcal{R})$

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

Proposition 7. $(a.\ell \Leftarrow \varsigma(x)b).\ell \approx (let\ x = (a.\ell \Leftarrow \varsigma(x)b)\ in\ b)$

The following laws characterise object constants and their interaction with the other constructs of the calculus.

Proposition 8. *Suppose* $o = [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..n}]$ *and* $j \in 1..n$.
(1) $o.\ell_j \approx (let\ x_j = o\ in\ b_j)$
(2) $(o.\ell_j \Leftarrow \varsigma(x)b) \approx [\ell_i = \varsigma(x_i)b_i\ ^{i \in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i\ ^{i \in j+1..n}]$
(3) $clone(o) \approx o$
(4) $(let\ x = o\ in\ \mathcal{R}[clone(x)]) \approx (let\ x = o\ in\ \mathcal{R}[o])$, *if* $x \notin fv(o)$
(5) $(let\ x = o\ in\ b) \approx b$, *if* $x \notin fv(b)$
(6) $(let\ x = a\ in\ let\ y = o\ in\ b) \approx (let\ y = o\ in\ let\ x = a\ in\ b)$,
if $x \notin fv(o)$ *and* $y \notin fv(a)$

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

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

From the laws for let and for object constants, the following calculation shows the validity of β_v-reduction, $(\lambda(x)b)(y) \approx b\{\!\{y/x\}\!\}$. Let $o = [arg = \varsigma(z)y, val = \varsigma(s)let\ x = s.arg\ in\ b]$ where $z \neq y$.

$$\begin{array}{rll}
(\lambda(x)b)(y) & \approx ((\lambda(x)b).arg \Leftarrow \varsigma(z)y).val & \text{by Prop. 6(1)} \\
& \approx o.val & \text{by Prop. 8(2) and } (\approx \text{Cong } \mathcal{R}) \\
& \approx let\ s = o\ in\ let\ x = s.arg\ in\ b & \text{by Prop. 8(1)} \\
& \approx let\ x = o.arg\ in\ b & \text{by Prop. 6(2)} \\
& \approx let\ x = (let\ z = o\ in\ y)\ in\ b & \text{by Prop. 8(1) and } (\approx \text{Cong } \mathcal{R}) \\
& \approx let\ x = y\ in\ b & \text{by Prop. 8(5) and } (\approx \text{Cong } \mathcal{R}) \\
& \approx b\{\!\{y/x\}\!\} & \text{by Prop. 6(1)}
\end{array}$$

This derivation uses the fact that operational equivalence is preserved by static reduction contexts, ($\approx$ Cong $\mathcal{R}$). More generally, to reason compositionally we need operational equivalence to be preserved by arbitrary term constructs, that is, to be a congruence. The following may be proved in several ways, most simply by an adaptation of the corresponding congruence proof for a λ-calculus with references by Honsell, Mason, Smith and Talcott [14].

Proposition 9. *Operational equivalence is a congruence.*

From Proposition 9 it easily follows that operational equivalence coincides with Morris-style contextual equivalence. Let a *term context*, $\mathcal{C}$, be a term containing some holes. Let the term $\mathcal{C}[a]$ be the outcome of filling each hole in the context $\mathcal{C}$ with the term a.

Theorem 10. $a \approx a'$ *if and only if for all term contexts* $\mathcal{C}$ *with* $locs(\mathcal{C}) = \varnothing$, $\mathcal{C}[a]$ *and* $\mathcal{C}[a']$ *are closed, that* $(\mathcal{C}[a], [])\Downarrow \Leftrightarrow (\mathcal{C}[a'], [])\Downarrow$.

Earlier studies of operational equivalence of stateless object calculi [10,15,24] rely on bisimulation equivalence. See Stark [26] for an account of the difficulties of defining bisimulation in the presence of imperative effects. The main influence on this section is the literature on operational theories for functional languages with state [14,18]. Agha, Mason, Smith and Talcott study contextual equivalence, but not CIU equivalence, for a concurrent object-oriented language based on actors [5]. Ours is the first development of CIU equivalence for an object-oriented language. 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.

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 [14] where it is possible to express such conditions directly.

5 Example: Static Resolution of Labels

In Section 3 we showed how to compile the imperative object calculus to an abstract machine that represents objects as finite lists of labels paired with method closures. 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 always compute statically the object to which a select or an update will apply. However, when the select or update is performed on a newly created object, or to self, it is possible to resolve method labels statically. The purpose of this section is to exercise our framework by presenting an algorithm for statically resolving method labels in these situations and proving it correct.

To represent our intermediate language, we begin by extending the syntax of terms to include selects of the form $a.j$ and updates of the form $a.j \Leftarrow \varsigma(x)b$, where j is a positive integer offset. The intention is that at runtime, a resolved select $\iota.j$ proceeds by running the jth method of the object stored at ι. If the jth method of this object has label ℓ, this will have the same effect as $\iota.\ell$. Similarly, an update $\iota.j \Leftarrow \varsigma(x)b$ proceeds by updating the jth method of the object stored at ι with method $\varsigma(x)b$. If the jth method of this object has label ℓ, this will have the same effect as $\iota.\ell \Leftarrow \varsigma(x)b$. 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 omit all the details. All the results proved in Sections 3 and 4 remain true for this extended language.

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

$A ::= [\ell_i \;^{i \in 1..n}]$	layout type (ℓ_i distinct)
$SE ::= [x_i \mapsto A_i \;^{i \in 1..n}]$	static environment (x_i distinct)

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 of the form $[\ell_i = \varsigma(x_i)b_i \;^{i \in 1..n}]$. 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 SE is a finite map that associates layout types to the free variables of a term.

We express the algorithm as the following recursive routine $\mathit{resolve}(SE, a)$, which takes an environment SE and a static term a with $\mathit{fv}(a) \subseteq \mathit{dom}(SE)$, 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.

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

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

$$[val = \varsigma(s)s.\mathit{ff}, tt = \varsigma(s)[], \mathit{ff} = \varsigma(s)[]]$$

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

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

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

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. We have proved that this optimisation is correct in the sense that a' is operationally equivalent to a.

Theorem 11. *Suppose a is a closed static term. If routine $resolve([], a)$ returns (a', A), then $a \approx a'$.*

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.

In general, there are many algorithms for optimising access to objects; see Chambers [7], 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 [21], 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.

6 Conclusions

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

The design of our object-oriented abstract machine was not particularly difficult; we simply extended Leroy's abstract machine with instructions for manipulating objects. Our first result is a correctness proof for the abstract machine and its compiler, Theorem 5. Such results are rather more difficult than proofs of interpretive abstract machines. Our contribution is a direct proof method which avoids the need for any metalanguage—such as a calculus of explicit substitutions. Our second result is that Mason and Talcott's CIU equivalence coincides with Morris-style contextual equivalence, Theorem 10. The benefit of CIU equivalence is that it allows the verification of compiler optimisations. We illustrate this by proving Theorem 11, which asserts that an optimisation algorithm from our implementation preserves contextual equivalence.

This is the first study of correctness of compilation to an object-oriented abstract machine. It is also the first study of program equivalence for the imperative object calculus, a topic left unexplored by Abadi and Cardelli's book. To the best of our knowledge, the only other work on the imperative object calculus is a program logic due to Abadi and Leino [4] and a brief presentation, without discussion of equivalence, of a labelled transition system for untyped imperative objects in the thesis of Andersen and Pedersen [6].

In principle, we believe our compiler correctness proof would scale up to proving correctness of a Java compiler emitting instructions for the Java virtual machine (JVM) [17]. To carry this out would require formal descriptions of the operational semantics of Java, the JVM and the compiler. Due to the scale of the task, the proof would require machine support.

Acknowledgements Martín Abadi, Carolyn Talcott and several anonymous referees commented on a draft. Gordon holds a Royal Society University Research Fellowship. Hankin holds an EPSRC Research Studentship. Lassen is supported by a grant from the Danish Natural Science Research Council.

References

1. M. Abadi and L. Cardelli. An imperative object calculus: Basic typing and soundness. In *Proceedings SIPL'95*, 1995. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign.
2. M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(13):151–166, 1996.
3. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
4. M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In *Proceedings TAPSOFT '97*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, April 1997.
5. G. Agha, I. Mason, S. Smith and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1), January 1997.
6. D.S. Andersen and L.H. Pedersen. An operational approach to the ς-calculus. Master's thesis, Department of Mathematics and Computer Science, Aalborg, 1996. Available as Report R–96–2034.
7. C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Computer Science Department, Stanford University, March 1992.
8. M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.
9. A.D. Gordon, S.B. Lassen and P.D. Hankin. Compilation and equivalence of imperative objects. Technical Report 429, University of Cambridge Computer Laboratory, 1997. Also appears as BRICS Report RS–97–19, BRICS, Department of Computer Science, University of Aarhus.
10. A.D. Gordon and G.D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings POPL'96*, pages 386–395. ACM, 1996. Accepted for publication in *Information and Computation*.

11. J.D. Guttman, V. Swarup and J. Ramsdell. The VLISP verified scheme system. *Lisp and Symbolic Computation*, 8(1/2):33–110, 1995.
12. J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 4(2):415–489, 1992.
13. T. Hardin, L. Maranget and B. Pagano. Functional back-ends and compilers within the lambda-sigma calculus. In *ICFP'96*, May 1996.
14. F. Honsell, I. Mason, S. Smith and C. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 1993.
15. H. Hüttel and J. Kleist. Objects as mobile processes. In *Proceedings MFPS'96*, 1996.
16. X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.
17. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1997.
18. I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.
19. E. Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1989. Earlier version in *Proceedings LICS'89*.
20. J.H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, December 1968.
21. A. Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings POPL'92*, pages 154–165. ACM, 1992.
22. G.D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
23. M. Rittri. *Proving compiler correctness by bisimulation*. PhD thesis, Chalmers, 1990.
24. D. Sangiorgi. An interpretation of typed objects into typed π-calculus. In *FOOL 3, New Brunswick*, 1996.
25. P. Sestoft. Deriving a lazy abstract machine. Technical Report 1994-146, Department of Computer Science, Technical University of Denmark, September 1994.
26. I. Stark. Names, equations, relations: Practical ways to reason about *new*. In *TLCA '97*, number 1210 in LNCS, pages 336–353. Springer, 1997.
27. C. Talcott. Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 347–390. Cambridge University Press, 1997. To appear.
28. M. Wand. Compiler correctness for parallel languages. In *Proceedings FPCA'95*, pages 120–134. ACM, June 1995.