Bisimilarity for a First-Order Calculus of Objects with Subtyping*

Andrew D. Gordon and Gareth D. Rees
University of Cambridge Computer Laboratory
http://www.cl.cam.ac.uk/users/{adg,gdr11}/

October 1995

Abstract

Bisimilarity (also known as 'applicative bisimulation') has attracted a good deal of attention as an operational equivalence for λ-calculi. It approximates or even equals Morris-style contextual equivalence and admits proofs of program equivalence via co-induction. It has an elementary construction from the operational definition of a language. We consider bisimilarity for one of the typed object calculi of Abadi and Cardelli. By defining a labelled transition system for the calculus in the style of Crole and Gordon and using a variation of Howe's method we establish two central results: that bisimilarity is a congruence, and that it equals contextual equivalence. So two objects are bisimilar iff no amount of programming can tell them apart. Our third contribution is to show that bisimilarity soundly models the equational theory of Abadi and Cardelli. This is the first study of contextual equivalence for an object calculus and the first application of Howe's method to subtyping. By these results, we intend to demonstrate that operational methods are a promising new direction for the foundations of object-oriented programming.

*To appear in *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 1996.*

1 Motivation

Abadi and Cardelli (1994a, 1994b, 1994c) present a number of related calculi that formalise aspects of object-oriented programming languages, including method update (the ability to modify the behaviour of an object by altering one of its methods) and object subsumption (the ability to emulate an object with an object that has more methods). They give equational theories for their calculi, present a denotational semantics based on partial equivalence relations for the largest calculus and show that the equational theory is sound. Their object calculi form an extremely simple yet clearly object-oriented setting in which to seek type systems that support styles of object-oriented programming found in full-blown languages. Hence they are an important subject of research.

Abadi and Cardelli's goal was to study type systems for objects by abandoning complex encodings of objects as λ-terms and to study primitive objects in their own right. Our goal here is to study operational equivalence of objects in its own right, instead of via denotational semantics, another kind of encoding.

We work with $\mathbf{Ob_{1<:\mu}}$ (Abadi and Cardelli 1994c), a first-order stateless object calculus including objects, recursive types and a ground type of Booleans. We take Morris-style contextual equivalence (Morris 1968) to be the natural operational equivalence on objects: two programs are equivalent unless there is a distinguishing context of ground type such that when each program is placed in the context, one converges but the other diverges. Following earlier work on functional languages (Crole and Gordon 1995; Gordon 1995), we define a CCS-style labelled transition system for the object calculus and replay the definition of (strong) bisimilarity from CCS. Our bisimilarity descends from a line of work on operational equivalence for functional calculi beginning with Abramsky's applicative bisimulation (Abramsky and Ong 1993). The new elements here are subtyping and objects. Using an extension of the method of Howe (1989) we prove Theorem 1, that bisimilarity is a congruence. Theorem 2, that bisimilarity equals contextual equivalence, then follows easily. The quantification over all contexts makes contextual equivalence hard to establish directly. The purpose of the labelled transition system, justified by Theorem 2, is to admit CCS-style bisimulation proofs of contextual equivalence. We use this style of proof—a form of co-induction—for our final result, Theorem 3, that bisimilarity soundly models all of Abadi and Cardelli's equational theory.

We briefly examine equivalences between two example objects suggested by Abadi and Cardelli. One of the equivalences follows by co-induction but not from their equational theory.

Our framework appears to be robust. Our main results continue to hold when we extend the pure object calculus with functions, records, variants and dynamic types. We leave a study of polymorphic types as future work, but see Rees (1994).

2 An object calculus

The expressions of the $\mathbf{Ob_{1<:\mu}}$ are *object formation*, $[\ell_1 = \varsigma(x_1{:}A_1)e_1, \ldots, \ell_n = \varsigma(x_n{:}A_n)e_n]$ (where an expression of the form $\varsigma(x_i{:}A_i)e_i$ is a *method*), *method selection*, $a.\ell$, and *method update*, $a.\ell \Leftarrow \varsigma(x{:}A)e$, together with constructs for Booleans and conditionals. We use the metavariables x and X for variables and type variables, and e and E for possibly open expressions and types respectively. Let I stand for finite indexing sets and ℓ for labels, drawn from some countably infinite set. Formally, the grammars of *types*, E, and *expressions*, e, are given as follows.

$$\begin{array}{rcl} E & ::= & X \mid \texttt{Top} \mid \texttt{Bool} \mid [\ell_i{:}E_i]_{i\in I} \mid \mu(X)E \\ e & ::= & x \mid [\ell_i = \varsigma(x_i{:}E_i)e_i]_{i\in I} \mid e.\ell \\ & & \mid e.\ell \Leftarrow \varsigma(x{:}E)e \mid \texttt{fold}(E,e) \\ & & \mid \texttt{unfold}(e) \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}(e,e,e) \end{array}$$

We identify expressions and types up to α-conversion, denoted by $\equiv$. We write $\phi[\psi/x]$ for the outcome of substituting phrase ψ for each occurrence of (type or expression) variable x in phrase ϕ.

An *environment*, Γ, is a finite list of assignments of closed types to variables, $x{:}A$, followed by a finite list of type variable bounds, $X <: E$. Let Γ, X, Γ' be short for $\Gamma, X <: \texttt{Top}, \Gamma'$. Let $Dom(\Gamma)$ be the set of variables and type variables bound or assigned in Γ. The static semantics of the calculus consists of five inductively defined judgments: $\Gamma \vdash \diamond$ (the environment Γ is well-formed), $E \succ Y$ (the type E is formally contractive in variable Y), $\Gamma \vdash E$ (the type expression E is well-formed), $\Gamma \vdash E <: E'$ (the type E is a subtype of E'), and $\Gamma \vdash e{:}A$ (the expression e has the closed type A). These judgments are given inductively by the rules in Tables 3, 4, 5, 6 and 7. We follow a metavariable convention based on the following sets.

$$\begin{array}{rcl} A, B \in \mathit{Type} & \stackrel{\text{def}}{=} & \{E \mid \varnothing \vdash E\} \\ a, b \in \mathit{Prog}(A) & \stackrel{\text{def}}{=} & \{e \mid \varnothing \vdash e : A\} \end{array}$$

By *program* we specifically mean closed expressions contained in $\mathit{Prog}(A)$ for some A, respectively. As usual we write $a{:}A$ to mean $\varnothing \vdash a : A$ and $A <: B$ for $\varnothing \vdash A <: B$.

Abadi and Cardelli discuss the $\mathbf{Ob_{1<:\mu}}$ type system at length. A few points are worth noting here. Numbers, functions, lists and so on can be encoded. Although the grammar allows the type annotations on ς-bound variables in an object to be distinct, if an object $[\ell_i = \varsigma(x_i{:}A_i)e_i]_{i\in I}$ has type B, then each $A_i <: B$ and in fact all the A_i are identical. The contractivity constraint on recursive types implies that any closed type can be decomposed into the form $\mu(X_1)\ldots\mu(X_n)E$ where E is one of $\texttt{Top}$, $\texttt{Bool}$ or $[\ell_i{:}E_i]_{i\in I}$. Whenever $A <: B$ and both A and B are object types, they must have the forms $[\ell_i{:}A_i]_{i\in I}$ and $[\ell_j{:}A_j]_{j\in J}$, respectively, with $J \subseteq I$. In other words, an object type is *invariant* in its component types, that is, neither covariant nor contravariant. This is necessary because methods have a contravariant dependence on self, and because they support both selection and update. Abadi and Cardelli (1994b, 1995) develop richer type systems that address this limitation, but we do not consider them here.

We need the following substitution and bound weakening lemmas, which are standard.

Lemma 1 *If* $\Gamma, x{:}A, \Gamma' \vdash e : B$ *and* $\Gamma \vdash e' : A$*, then* $\Gamma, \Gamma' \vdash e[e'/x] : B$.

Lemma 2 *If* $\Gamma, x{:}A, \Gamma' \vdash e : B$ *and* $A' <: A$ *then* $\Gamma, x{:}A', \Gamma' \vdash e : B$ *too.*

Abadi and Cardelli present a many-step deterministic evaluation relation, $\leadsto$, for $\mathbf{Ob}_{1<:\mu}$. For our purposes, it is more convenient to reformulate it as a single-step reduction relation. To specify the evaluation strategy we need the following notion of a context. Let '$-$' be a distinguished variable used to stand for a *hole* in a program. Let a *context* be an expression e such that the only free variable, if any, is $-$. If e is a context and a is a program, we write $e[a]$ short for the program $e[a/-]$. These are not variable-capturing contexts.

For each type A define the set $\mathit{Value}(A) \subseteq \mathit{Prog}(A)$ (with typical members u, v) of *values* of the following forms.

$$[\ell_i = \varsigma(x_i{:}B)e_i]_{i \in I} \quad \mathtt{fold}(B, v) \quad \mathtt{true} \quad \mathtt{false}$$

Let *Value* be the set of values at any type, that is,

$$\mathit{Value} = \bigcup \{\mathit{Value}(A) \mid A \in \mathit{Type}\}.$$

The notation $a \mapsto b$ means that a reduces to b in a single step of reduction; it is defined inductively by the following axiom schemes

(Red Select) $a.\ell_j \mapsto e_j[a/x_j]$
where $a \equiv [\ell_i = \varsigma(x_i{:}A_i)e_i]_{i \in I}$ and $j \in I$.

(Red Update) $a.\ell_j \Leftarrow \varsigma(x{:}B)e \mapsto a'$
where $a \equiv [\ell_i = \varsigma(x_i{:}A_i)e_i]_{i \in I}$,
$a' \equiv [\ell_j = \varsigma(x{:}A_j)e, \ell_i = \varsigma(x_i{:}A_i)e_i]_{i \in I - \{j\}}$
and $j \in I$.

(Red Unfold) $\mathtt{unfold}(\mathtt{fold}(A, v)) \mapsto v$

(Red If True) $\mathtt{if}(\mathtt{true}, a_1, a_2) \mapsto a_1$

(Red If False) $\mathtt{if}(\mathtt{false}, a_1, a_2) \mapsto a_2$

with an evaluation strategy given by

$$\frac{a \mapsto b}{\mathcal{E}[a] \mapsto \mathcal{E}[b]} \text{ (Red Experiment)}$$

where an *experiment*, $\mathcal{E}$, is a context with one hole, of one of the following forms.

$$-.\ell \quad -.\ell \Leftarrow \varsigma(x{:}A)e$$
$$\mathtt{unfold}(-) \quad \mathtt{fold}(A, -) \quad \mathtt{if}(-, a_1, a_2)$$

The relation $\mapsto$ is weak in the sense that it is closed only under experiments, not arbitrary contexts. There is no experiment to allow reduction under ς-binders. (Red Update) preserves the property that whenever an object $[\ell_i = \varsigma(x_i{:}A_i)e_i]_{i \in I}$ has type A, each $A_i <: A$ and indeed all the A_i's are identical.

Experiments are just atomic evaluation contexts (Felleisen and Friedman 1986); every program can be decomposed uniquely as follows.

Lemma 3 *If* $a{:}A$ *there is a unique list of experiments* $\mathcal{E}_1, \ldots, \mathcal{E}_n$, $n \geq 0$, *and value* v *such that* $a \equiv \mathcal{E}_1[\ldots \mathcal{E}_n[v] \ldots]$.

Lemma 4 *The values are the normal forms of* $\mapsto$*, that is, whenever* a *is a program,* $a \in \mathit{Value}$ *iff* $\neg \exists b (a \mapsto b)$.

The reduction rules are deliberately type independent, in the sense that although they manipulate type information contained in programs, they are not contingent on the form of the types they manipulate. For instance, (Red Update) allows for the type bounds, A_i, to be distinct although we know them to be identical. Hence ill-typed expressions can be reduced. This is a redundancy in the dynamic semantics, but is harmless because we are only interested in statically typable programs.

We can easily prove determinacy and subject reduction.

Lemma 5 *If* $a \mapsto b$ *and* $a \mapsto c$*, then* $b \equiv c$.

Lemma 6 *If* $a{:}A$ *and* $a \mapsto b$ *then* $b{:}A$.

As usual, let the relation $\mapsto^*$ be the reflexive and transitive closure of $\mapsto$. We now recover a many-step evaluation relation, $\Downarrow$, and three standard predicates as follows.

$$\begin{array}{rcll} a \mapsto & \stackrel{\text{def}}{=} & \exists b(a \mapsto b) & \text{‘}a\text{ reduces'} \\ a \Downarrow b & \stackrel{\text{def}}{=} & a \mapsto^* b \,\&\, \neg(b \mapsto) & \text{‘}a\text{ evaluates to }b\text{'} \\ a\Downarrow & \stackrel{\text{def}}{=} & \exists b(a \Downarrow b) & \text{‘}a\text{ converges'} \\ a\Uparrow & \stackrel{\text{def}}{=} & \forall b(a \mapsto^* b \Rightarrow b \mapsto) & \text{‘}a\text{ diverges'} \end{array}$$

We have $a\Downarrow$ iff not $a\Uparrow$. We have recovered the many-step evaluation relation $\leadsto$; $a \Downarrow b$ iff $\vdash a \leadsto b$. The proof is standard.

We introduced $\mathbf{Ob_{1<:\mu}}$ without recursive programs, but we can define them using objects; for example, let $\mu(x{:}A)e$ abbreviate the expression $[\ell = \varsigma(s{:}[\ell{:}A])e[{}^{s.\ell}\!/\!{}_x]].\ell$. We have $\mu(x{:}A)e \mapsto e[{}^{\mu(x:A)e}\!/\!{}_x]$. Hence there is a divergent program at every type. Let Ω^A be $\mu(x{:}A)x$; $\Omega^A \mapsto \Omega^A$ so $\Omega^A\Uparrow$. The definability of Ω^A and the contractivity condition on recursive types imply the following lemma.

Lemma 7 $\forall A \in \mathit{Type}\, \exists a_1, a_2{:}A\ (a_1\Downarrow \,\&\, a_2\Uparrow)$.

The results of this paper would hold for the language without contractivity, but its presence simplifies the statement of certain results, because contractivity rules out types with no values, such as $\mu(X)X$.

3 Contextual equivalence

We take Morris' contextual equivalence (Morris 1968; Plotkin 1977), also known as ‘observational congruence' (Meyer and Cosmadakis 1988), to be the natural operational equivalence on objects. First we introduce the idea of a relation between expressions of matching types. Let a *proved program* be a pair a_A such that $a{:}A$. Let P and Q range over proved programs. Let *Rel* be the universal relation on proved programs of the same type, given as follows.

$$\mathit{Rel} \stackrel{\text{def}}{=} \{(a_A, b_A) \mid a{:}A \,\&\, b{:}A\}$$

We use $\mathcal{R}$ and $\mathcal{S}$ for subsets of *Rel*, that is, relations on proved programs that respect typing. If $\mathcal{R} \subseteq \mathit{Rel}$, then for any type A, define $\mathcal{R}_A = \{(a, b) \mid (a_A, b_A) \in \mathcal{R}\}$. So the notation ‘$a \,\mathcal{R}_A\, b$' means that $(a_A, b_A) \in \mathcal{R}$.

Instead of using relations on proved programs, we could have used binary relations on expressions indexed by types. We use proved programs because when defining bisimilarity we can work simply in the complete lattice of subsets of *Rel* ordered by $\subseteq$, rather than of indexed sets.

For each closed type A, let *A-contextual equivalence*, $\stackrel{A}{\simeq} \subseteq \mathit{Rel}$, be the relation on proved programs given by the following.

$$a \stackrel{A}{\simeq}_B b \text{ iff whenever } -{:}B \vdash e : A,\ e[a] \Downarrow \text{ iff } e[b] \Downarrow.$$

In other words, two programs are A-contextually equivalent iff their termination behaviour is the same whenever they are placed in a larger program, e, of type A.

Proposition 8

(1) *Suppose there exists a context e such that $-{:}A \vdash e : B$ and for all programs $a{:}A$, we have $a\Downarrow$ iff $e[a]\Downarrow$. Then $\stackrel{B}{\simeq} \subseteq \stackrel{A}{\simeq}$.*

(2) *If $A <: B$ then $\stackrel{B}{\simeq} \subseteq \stackrel{A}{\simeq}$.*

(3) *For all A, $\stackrel{\mathtt{Top}}{\simeq} \subseteq \stackrel{A}{\simeq}$.*

(4) *For all B, $\stackrel{B}{\simeq} \subseteq \stackrel{\mathtt{Bool}}{\simeq}$.*

(5) $\stackrel{\mathtt{Top}}{\simeq} \neq \stackrel{\mathtt{Bool}}{\simeq}$.

Proof

(1) Suppose for some some type C and programs a and b, we have $a \stackrel{B}{\simeq}_C b$. We must prove that $a \stackrel{A}{\simeq}_C b$. Suppose for some context e' that $-{:}C \vdash e'{:}A$. By symmetry, it is enough to show that if $e'[a]\Downarrow$ then $e'[b]\Downarrow$. Consider the context $e[{}^{e'}\!/\!{}_-]$ satisfying $-{:}C \vdash e[{}^{e'}\!/\!{}_-] : B$, where e is as given in the statement of the proposition. Then $e[e'[a]]\Downarrow$, and since $a \stackrel{B}{\simeq}_C b$, we have $e[e'[b]]\Downarrow$; hence $e'[b]\Downarrow$ as required.

(2) A corollary of part (1), taking $e \equiv -$ and using subsumption.

(3) A corollary of part (2), since $A <: \mathtt{Top}$.

(4) A corollary of part (1), taking $e \equiv \mathtt{if}(-, v_B, v_B)$ for some value v_B of type B (we know such a value must exist by Lemma 7).

(5) We have $\neg(\mathtt{true} \overset{\mathtt{Top}}{\simeq}_{\mathtt{Top}} \Omega^{\mathtt{Top}})$ but $\mathtt{true} \overset{\mathtt{Bool}}{\simeq}_{\mathtt{Top}} \Omega^{\mathtt{Top}}$. The former is immediate; the latter is trivial once we have Theorem 2. ■

Of this family of equivalence relations, $\overset{\mathtt{Top}}{\simeq}$ makes the most distinctions between programs, and $\overset{\mathtt{Bool}}{\simeq}$ the fewest. The only substantial difference among the relations is the set of types at which termination is distinguishable from non-termination.

In a language with call-by-value functions, we can construct a context satisfying part (1) for any two types A and B, namely the application $((\lambda(x{:}A)v_B) -)$ where v_B is some value of type B. Hence, in a call-by-value language, all the $\overset{A}{\simeq}$'s are equal.

We choose to take $\overset{\mathtt{Bool}}{\simeq}$ as our notion of operational equivalence for this study for three reasons. First, it is the notion of contextual equivalence used by Plotkin (1977) and is standard in studies of the language PCF. Second, it is the most generous of the A-contextual equivalences. Third, since one of our motivations is to validate the equational theory of $\mathbf{Ob_{1<:\mu}}$, we must choose a contextual equivalence in which their equations are sound. In particular, the rule

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash e' : B}{\Gamma \vdash e \leftrightarrow e' : \mathtt{Top}} \text{ (Eq Top)}$$

must hold, so $\overset{\mathtt{Top}}{\simeq}$ is inappropriate, because it allows the observation of termination behaviour at type Top. Hence it distinguishes true and Ω at type Top, because the former converges but the latter diverges. We will see later that (Eq Top) holds for $\overset{\mathtt{Bool}}{\simeq}$.

Another candidate was $\overset{[]}{\simeq}$, which does satisfy (Eq Top) and would be the most natural choice in $\mathbf{Ob_{1<:\mu}}$ without Booleans. It is finer-grained than $\overset{\mathtt{Bool}}{\simeq}$; it distinguishes $[]$ and $\Omega^{[]}$ whereas $\overset{\mathtt{Bool}}{\simeq}$ identifies them.

We will define *contextual equivalence*, $\simeq \subseteq Rel$, to stand for $\overset{\mathtt{Bool}}{\simeq}$, and define *contextual order*, $\sqsubseteq\!\!\!\sim \subseteq Rel$, as follows.

$$a \mathrel{\underset{\sim}{\sqsubset}}_A b \quad \text{iff} \quad \text{whenever } -{:}A \vdash e : \mathtt{Bool}, \; e[a] \Downarrow \text{ implies } e[b] \Downarrow \text{ too.}$$

Note that $a \simeq_A b$ iff $a \mathrel{\underset{\sim}{\sqsubset}}_A b$ and $b \mathrel{\underset{\sim}{\sqsubset}}_A a$.

It is easy to show two programs are contextually distinct: just exhibit a single context that tells them apart. But to show equivalence requires a quantification over all contexts. The point of the next section is to characterise contextual equivalence co-inductively as a kind of bisimilarity, and hence to admit CCS-style bisimulation proofs of equivalence. Contextual equivalence is defined in terms of one-off tests consisting of composite contexts; bisimilarity is defined in terms of multiple atomic observations on objects. When proving programs equal it is often easier to consider a series of atomic observations rather than all possible contexts.

4 Bisimilarity

4.1 Labelled transitions

We define a labelled transition system that characterises the atomic observations one can make of a proved program. The notation $P \xrightarrow{\alpha} Q$ means that the proved program P does an *action* α to become another proved program Q. As usual we write $P \xrightarrow{\alpha}$ mean that there is some Q with $P \xrightarrow{\alpha} Q$.

The simplest labelled transition system to characterise contextual equivalence co-inductively is the following.

$$\frac{-{:}A \vdash \mathcal{E} : B}{a_A \xrightarrow{\mathcal{E}} \mathcal{E}[a]_B} \text{ (Trans Exper)}$$

$$\frac{a\Downarrow}{a_{\mathtt{Bool}} \xrightarrow{\mathtt{val}} \mathbf{0}} \text{ (Trans Val)}$$

where an action is either an experiment, $\mathcal{E}$, or val, and $\mathbf{0}$ is disjoint from the set of programs. We could prove that CCS-style bisimilarity according to this labelled transition system equals contextual equivalence. This is a direct generalisation of Milner's context lemma for PCF (Milner 1977). We can do better than this by describing a labelled transition system in which there are fewer transitions available to proved programs: this reduction in size will simplify some of our proofs.

We divide the types of $\mathbf{Ob_{1<:\mu}}$ into two classes, *active* and *passive*. Only Bool is active. Recursive types, object types and Top are passive. At active types a program must converge to a value

$$\frac{a \Downarrow v \in \{\texttt{true}, \texttt{false}\}}{a_{\texttt{Bool}} \xrightarrow{v} \mathbf{0}} \text{(Trans Bool)} \qquad \frac{A \equiv [\ell_i{:}A_i]_{i \in I} \quad j \in I}{a_A \xrightarrow{\ell_j} a.\ell_{j\,A_j}} \text{(Trans Select)}$$

$$\frac{A \equiv [\ell_i{:}A_i]_{i \in I} \quad j \in I \quad x{:}A \vdash e : A_j}{a_A \xrightarrow{\ell_j \Leftarrow \varsigma(x)e} a.\ell_j \Leftarrow \varsigma(x{:}A)e_A} \text{(Trans Update)} \qquad \frac{A \equiv \mu(X)E \quad B \equiv E[A/X]}{a_A \xrightarrow{\texttt{unfold}} \texttt{unfold}(a)_B} \text{(Trans Unfold)}$$

Table 1: Rules of the labelled transition system

before it can be observed; at passive types a program does actions unconditionally, whether or not it converges. The observable actions, $\alpha \in Act$, take the following forms.

$$\texttt{true} \quad \texttt{false} \quad \ell \quad \ell \Leftarrow \varsigma(x)e \quad \texttt{unfold}$$

These actions correspond to the actions of the labelled transition system based on (Trans Exper) and (Trans Val), except that `true` and `false` replace actions of the form $\texttt{if}(-, a, b)$. We have erased type annotations from the update actions because they contain redundant information that will in any case be erased by the (Red Update) rule.

The labelled transition system we shall work with is the family of relations $(\xrightarrow{\alpha} \mid \alpha \in Act)$ given by the rules in Table 1, such that whenever $P \xrightarrow{\alpha} Q$, each of P and Q is a proved program. Let $\mathbf{0}$ be $a_{\texttt{Top}}$, for some arbitrary program $a{:}\texttt{Top}$. The purpose of $\mathbf{0}$ is that it has no actions; after observing ground data there is nothing more to observe.

We can characterise the observable actions at each type. For each type A, define the set $Act(A) \subseteq Act$ as follows.

$$\begin{aligned} Act(\texttt{Top}) &= \{\} \\ Act(\texttt{Bool}) &= \{\texttt{true}, \texttt{false}\} \\ Act(\mu(X)E) &= \{\texttt{unfold}\} \\ Act([\ell_i{:}A_i]_{i \in I}) &= \{\ell_i, \ell_i \Leftarrow \varsigma(x)e \mid i \in I \\ &\qquad \& \ x{:}[\ell_i{:}A_i]_{i \in I} \vdash e : A_i\} \end{aligned}$$

Lemma 9 *$\alpha \in Act(A)$ iff $\exists a{:}A(a_A \xrightarrow{\alpha})$.*

We can make more observations at a subtype than a supertype.

Lemma 10 *If $A <: B$ then $Act(B) \subseteq Act(A)$.*

The following is a trivial fact for $\mathbf{Ob_{1<:\mu}}$, as `Bool` is the only active type, but it holds in the extensions of $\mathbf{Ob_{1<:\mu}}$ we have considered, in which more types are active.

Lemma 11 *If A is active then $a_A \xrightarrow{\alpha} b_B$ iff $\exists v \in Value(a \Downarrow v \ \& \ v_A \xrightarrow{\alpha} b_B)$.*

Our labelled transition system is image-singular, in the following sense.

Lemma 12 *If $P \xrightarrow{\alpha} Q$ and $P \xrightarrow{\alpha} Q'$, then $Q \equiv Q'$.*

Because of subsumption, the system based on (Trans Exper) and (Trans Val) is not image-singular. For example, the following two transitions are derivable from (Trans Exper).

$$\begin{aligned} a_{[\ell:\texttt{Bool}]} &\xrightarrow{-.\ell} a.\ell_{\texttt{Bool}} \\ a_{[\ell:\texttt{Bool}]} &\xrightarrow{-.\ell} a.\ell_{\texttt{Top}} \end{aligned}$$

4.2 Definition of bisimilarity

The *derivation tree* of a proved program P is the potentially infinite tree whose nodes are proved programs, whose arcs are labelled transitions, and which is rooted at P. Following Milner (1989), we wish to regard two proved programs as behaviourally equivalent iff their derivation trees are the same when we ignore the syntactic structure of the programs labelling the nodes and the ordering of the arcs from each node. We formalise this idea in the standard way. First define two functions $[-], \langle - \rangle : \wp(Rel) \to \wp(Rel)$ by

$$\begin{aligned} [\mathcal{S}] &\stackrel{\text{def}}{=} \{(P, Q) \mid \text{whenever } P \xrightarrow{\alpha} P' \text{ there is } Q' \\ &\qquad \text{with } Q \xrightarrow{\alpha} Q' \text{ and } P' \mathrel{\mathcal{S}} Q'\} \\ \langle \mathcal{S} \rangle &\stackrel{\text{def}}{=} [\mathcal{S}] \cap [\mathcal{S}^{\text{op}}]^{\text{op}} \end{aligned}$$

where $\mathcal{R}^{\mathrm{op}} = \{(b,a) \mid (a,b) \in \mathcal{R}\}$ for any binary relation $\mathcal{R}$. These are both monotone functions on $\wp(Rel)$. Let a relation $\mathcal{S} \subseteq Rel$ be a *bisimulation* iff $\mathcal{S} \subseteq \langle\mathcal{S}\rangle$. Let *bisimilarity*, $\sim \subseteq Rel$, be the union of all the bisimulations. By the Tarski–Knaster theorem, bisimilarity is the greatest fixpoint of $\langle - \rangle$. In other words, bisimilarity is the greatest relation to satisfy the following: whenever $(P,Q) \in Rel$, $P \sim Q$ iff

(1) $P \xrightarrow{\alpha} P' \Rightarrow \exists Q'(Q \xrightarrow{\alpha} Q' \;\&\; P' \sim Q')$

(2) $Q \xrightarrow{\alpha} Q' \Rightarrow \exists P'(P \xrightarrow{\alpha} P' \;\&\; P' \sim Q')$.

If $\mathcal{S}$ is a bisimulation, $\mathcal{S} \subseteq \sim$ by definition of $\sim$; this is the co-induction principle associated with bisimilarity.

We shall need the preorder form of bisimilarity. Let relation $\mathcal{S} \subseteq Rel$ be a *simulation* iff $\mathcal{S} \subseteq [\mathcal{S}]$. *Similarity*, $\lesssim \subseteq Rel$, is the greatest fixpoint of $[-]$, that is, the union of all simulations.

4.3 Basic properties of bisimilarity

We can easily establish the following using co-induction.

Proposition 13

(1) $\lesssim$ *is a preorder and* $\sim$ *an equivalence relation.*

(2) $\sim = \lesssim \cap \lesssim^{\mathrm{op}}$.

Part (2) depends on image-singularity, Lemma 12. This property fails in a nondeterministic calculus such as CCS.

4.4 Bisimilarity and subtyping

Whenever $A <: B$, we would expect that if two programs are equal at the subtype, A, that they will be equal at the supertype, B. To prove this, we need the following lemma.

Lemma 14 *Relation* $\{(a_B, b_B) \mid \exists A(\varnothing \vdash A <: B \;\&\; a \sim_A b)\}$ *is a simulation.*

The following is a simple corollary by co-induction.

Proposition 15

(1) *If* $a \lesssim_A b$ *and* $A <: B$ *then* $a \lesssim_B b$.

(2) *If* $a \sim_A b$ *and* $A <: B$ *then* $a \sim_B b$.

In the following section we introduce the idea of a relation being a congruence, and in the next we prove that bisimilarity is one.

4.5 Congruence and precongruence

A congruence is an equivalence that is preserved by all contexts. To state this formally we must begin with a few preliminary definitions. Let a *substitution* be a function $\vec{\sigma} = {}^{a_1}\!/\!_{x_1}, \ldots, {}^{a_n}\!/\!_{x_n}$ $(n \geq 0)$ from expressions to expressions, which substitutes programs for free variables. The application of a substitution $\vec{\sigma}$ to an expression e is written $e[\vec{\sigma}]$. A substitution $\vec{\sigma} \equiv {}^{a_1}\!/\!_{x_1}, \ldots, {}^{a_n}\!/\!_{x_n}$ is a Γ-*closure* for an environment $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ iff each $a_i{:}A_i$. Let a *proved expression* be a triple (Γ, e, A) such that Γ is 'closed' (it contains no type variable bounds; that is, it only contains assignments of closed types to program variables), $A \in Type$ and $\Gamma \vdash e : A$. If the relation $\mathcal{R} \subseteq Rel$ then its *open extension*, $\mathcal{R}^\circ$, is the relation on proved expressions such that $(\Gamma, e, A)\; \mathcal{R}^\circ\; (\Gamma', e', A')$ iff $A \equiv A'$, $\Gamma \equiv \Gamma'$ and $e[\vec{\sigma}]\; \mathcal{R}\; e'[\vec{\sigma}]$ for all Γ-closures $\vec{\sigma}$. Open extension is a monotonic operator on relations between programs.

Lemma 16 *If* $\mathcal{R} \subseteq \mathcal{S}$ *then* $\mathcal{R}^\circ \subseteq \mathcal{S}^\circ$.

For instance, Rel° is the universal relation on pairs of proved expressions with matching types and environments. As a notational convention, if $\mathcal{R} \subseteq Rel^\circ$ we write $\Gamma \vdash e\; \mathcal{R}\; e' : A$ to mean that $((\Gamma, e, A), (\Gamma, e', A)) \in \mathcal{R}$.

If $\mathcal{R} \subseteq Rel^\circ$ then its *compatible refinement* (Gordon 1994) is the relation $\widehat{\mathcal{R}} \subseteq Rel^\circ$ that relates two expressions if they share the same outermost syntactic constructor, and their immediate subexpressions are pairwise related by $\mathcal{R}$. We say a relation $\mathcal{R} \subseteq Rel^\circ$ is a *precongruence* iff it contains its own compatible refinement, that is, $\widehat{\mathcal{R}} \subseteq \mathcal{R}$, and it satisfies the following rule, with $\leftrightarrow$ equal to $\mathcal{R}$.

$$\frac{\Gamma \vdash e \leftrightarrow e' : A \quad A <: B}{\Gamma \vdash e \leftrightarrow e' : B} \text{ (Eq Subsum)}$$

This definition of precongruence can easily be shown equivalent to a more conventional one based on substitution into variable-capturing contexts. If, in addition, a precongruence is an equivalence relation, we say it is a *congruence*.

4.6 Bisimilarity is a congruence

We will show that the open extension of bisimilarity is a congruence. Since bisimilarity is the symmetrisation of similarity, Proposition 13(2), it is enough

$$\frac{\begin{array}{c}A' \equiv [\ell_i{:}B_i]_{i\in I} \quad A'' \equiv [\ell_i{:}B_i, \ell_j{:}B_j]_{i\in I, j\in J} \\ I \cap J = \varnothing \quad \Gamma, x_i{:}A' \vdash e_i : B_i \ (i \in I) \quad \Gamma, x_j{:}A'' \vdash e_j : B_j \ (j \in J)\end{array}}{\Gamma \vdash [\ell_i = \varsigma(x_i{:}A')e_i]_{i\in I} \leftrightarrow [\ell_i = \varsigma(x_i{:}A'')e_i]_{i\in I\cup J} : A'} \text{(Eq Sub Object)}$$

$$\frac{A \equiv \mu(X)E \quad \Gamma \vdash e : A}{\Gamma \vdash \mathtt{fold}(A, \mathtt{unfold}(e)) \leftrightarrow e : A} \text{(Eval Fold)}$$

Table 2: Fragment of the equational theory of $\mathbf{Ob_{1<:\mu}}$

to prove that similarity is a precongruence. We do so using a form of Howe's method (1989). We define an auxiliary relation, $\lesssim^\bullet$, which by definition is a precongruence, and prove that $\lesssim^\bullet = \lesssim^\circ$. Let the *precongruence candidate*, $\lesssim^\bullet \subseteq \mathit{Rel}^\circ$, be the least relation closed under the following rule.

$$\frac{\begin{array}{c}\Gamma \vdash e \mathrel{\widehat{\lesssim^\bullet}} e'' : A' \quad A' <: A \\ \Gamma \vdash e'' \lesssim^\circ e' : A\end{array}}{\Gamma \vdash e \lesssim^\bullet e' : A} \text{(Cand Def)}$$

Since $\lesssim^\bullet$ is defined by exactly one rule, it is valid upwards, that is, whenever $\Gamma \vdash e \lesssim^\bullet e' : A$, there is some type $A' <: A$ and some expression e'' with $\Gamma \vdash e \mathrel{\widehat{\lesssim^\bullet}} e'' : A'$ and also $\Gamma \vdash e'' \lesssim^\circ e' : A$. We can easily prove the following properties of $\lesssim^\bullet$ by standard methods.

Lemma 17 *Relation $\lesssim^\bullet$ is reflexive, and the following rules are valid.*

$$\frac{\Gamma \vdash e \lesssim^\circ e' : A}{\Gamma \vdash e \lesssim^\bullet e' : A} \text{(Cand Sim)}$$

$$\frac{\Gamma \vdash e \mathrel{\widehat{\lesssim^\bullet}} e' : A}{\Gamma \vdash e \lesssim^\bullet e' : A} \text{(Cand Comp)}$$

$$\frac{\Gamma \vdash e \lesssim^\bullet e'' : A \quad \Gamma \vdash e'' \lesssim^\circ e' : A}{\Gamma \vdash e \lesssim^\bullet e' : A} \text{(Cand Right)}$$

$$\frac{\begin{array}{c}\Gamma, x{:}B \vdash e_1 \lesssim^\bullet e_1' : A \\ \Gamma \vdash e_2 \lesssim^\bullet e_2' : B\end{array}}{\Gamma \vdash e_1[e_2/x] \lesssim^\bullet e_1'[e_2'/x] : A} \text{(Cand Subst)}$$

Moreover, $\lesssim^\bullet$ is the least relation closed under the rules (Cand Comp), (Cand Right) *and* (Eq Subsum).

Unlike in previous applications of Howe's method, we need to relate $\lesssim^\bullet$ and subtyping. Given Lemma 14 the following new properties are easy to prove.

Lemma 18 *Both* (Eq Subsum) *and the rule*

$$\frac{A <: B \quad \Gamma, x{:}B, \Gamma' \vdash e \leftrightarrow e' : C}{\Gamma, x{:}A, \Gamma' \vdash e \leftrightarrow e' : C} \text{(Eq Asm Subsum)}$$

hold for $\leftrightarrow$ equal to $\lesssim^\circ$ and to $\lesssim^\bullet$.

The following lemma is the heart of the precongruence proof. The proof is detailed but follows the standard pattern.

Lemma 19 *Relation $\mathcal{S} = \{(a_A, a'_A) \mid \varnothing \vdash a \lesssim^\bullet a' : A\}$ is a simulation.*

Theorem 1 *Relation $\sim^\circ$ is a congruence.*

Proof By Lemma 19, $\mathcal{S}$ is a simulation, and hence $\mathcal{S} \subseteq \lesssim$ by co-induction. Open extension is monotone, Lemma 16, so $\mathcal{S}^\circ \subseteq \lesssim^\circ$. Now $\lesssim^\bullet \subseteq \mathcal{S}^\circ$ follows by (Cand Subst) and the reflexivity of $\lesssim^\bullet$. Hence we have $\lesssim^\bullet \subseteq \lesssim^\circ$. But (Cand Sim) provides the reverse inclusion, so in fact $\lesssim^\bullet = \lesssim^\circ$ and hence $\lesssim^\circ$ is a precongruence. By appeal to Proposition 13(2), $\sim^\circ$ is a congruence. ■

4.7 Bisimilarity equals contextual equivalence

The proof of our main result, Theorem 2, follows the standard pattern.

Lemma 20 *Both $\lesssim \subseteq \sqsubseteq$ and $\sim \subseteq \simeq$.*

Lemma 21 *Contextual order, $\sqsubseteq$, is a simulation.*

Theorem 2 $\sim = \simeq$.

Proof Apply co-induction to Lemma 21 and combine with Lemma 20. ■

5 Operational adequacy

The relationship between operational semantics and equivalence in $\mathbf{Ob_{1<:\mu}}$ is subtle. The following facts are straightforward to state and prove.

Proposition 22 *For any type A,*

(1) $\forall a, b{:}A(a \mapsto b \Rightarrow a \sim_A b)$;

(2) $\forall a, v{:}A(a \Downarrow v \Rightarrow a \sim_A v)$;

(3) $\forall a{:}A(a\Uparrow \Rightarrow a \sim_A \Omega^A)$.

Part (3) strengthened from '$\Rightarrow$' to 'iff' is an important property of equality and divergence. In the setting of the equality induced by a denotational semantics it is usually known as *computational adequacy* (see Pitts (1994), for instance). It does not hold at all types in $\mathbf{Ob_{1<:\mu}}$. Consider type `Top`. Any two programs at type `Top` are bisimilar, because there are no transitions at type `Top`. So `true` $\sim_{\texttt{Top}} \Omega$ but not `true`$\Uparrow$.

`Top` is an example of a *singular* type, one in which all programs are equal. Let 'E singular' be the least predicate on types to satisfy the following: `Top` singular; X singular for any variable X; $\mu(X)E$ singular if E singular; and $[\ell_i{:}E_i]_{i\in I}$ singular if each E_i singular.

A type is singular if neither it nor any of its subexpressions is `Bool`. So the type $\mu(X)[\ell_1{:}\texttt{Top}, \ell_2{:}X]$ is singular but the type $\mu(X)[\ell_1{:}\texttt{Bool}, \ell_2{:}X]$ is not. Intuitively, all programs are equal at a singular type because there are no `Bool`-contexts to tell them apart.

Proposition 23 *For any A,*

(1) *A singular iff* $\forall a, b{:}A(a \sim_A b)$;

(2) *A not singular iff* $\forall a{:}A(a \sim_A \Omega^A \Rightarrow a\Uparrow)$.

The proof, which we omit, uses the fact that bisimilarity equals contextual equivalence. Computational adequacy, that a program equals Ω iff it diverges, holds just at the non-singular types.

6 Validating the equational theory

Abadi and Cardelli (1994c) present an equational theory for $\mathbf{Ob_{1<:\mu}}$. Their relation is essentially the relation $\leftrightarrow \subseteq Rel^\circ$ that is inductively defined by the rules in Table 2, together with rules of equivalence, compatibility, closure under evaluation, (Eq Top) and (Eq Subsum). We can show that the open extension of bisimilarity is closed under the relevant equational rules and hence $\leftrightarrow \subseteq \sim^\circ$.

Most of the $\mathbf{Ob_{1<:\mu}}$ equational theory is easy to verify. The equivalence rules follow for bisimilarity from Proposition 13(1). The congruence rules follow directly from Theorem 1. The evaluation rules follow from Proposition 22. (Eq Top) follows from Proposition 23. (Eq Subsum) follows from Lemma 18. (Eval Fold) can easily be proved by co-induction. (Eq Sub Object) appears to be most easily proved via a direct proof that the two objects are contextually equivalent. We omit the proof. Since $\sim^\circ$ is closed under all the rules inductively defining the equational theory, it is sound in the following sense.

Theorem 3 $\leftrightarrow \subseteq \sim^\circ$.

Bisimilarity is no panacea—witness the direct proof of (Eq Sub Object)—but the bisimulation proofs of most of the equational rules would appear to be simpler than direct proofs of contextual equivalence.

The reverse inclusion does not hold; see Proposition 26 below for an example. In any case since the calculus presented here is Turing-powerful, no recursively enumerable equational theory such as $\leftrightarrow$ could be complete for operational equivalence.

7 Example

To demonstrate the power of the co-inductive proof principle, we consider an example, given in section 4.3 of Abadi and Cardelli (1994c) that does not follow from the equational theory. A *field* is a degenerate method that does not depend on its self parameter. Let $e'.\ell := e$ be short for $e'.\ell \Leftarrow \varsigma(x{:}A)e$ for some type A such that $e'{:}A$, and $[\ell = e, \ldots]$ be short for $[\ell = \varsigma(x{:}A)e, \ldots]$, for some x not in the free variables of e. Define a type A and two objects a and b as follows.

$$\begin{aligned} A &\stackrel{\text{def}}{=} [x{:}\texttt{Bool}, f{:}\texttt{Bool}] \\ a{:}A &\stackrel{\text{def}}{=} [x = \texttt{true}, f = \texttt{true}] \\ b{:}A &\stackrel{\text{def}}{=} [x = \texttt{true}, f = \varsigma(s{:}A)s.x] \end{aligned}$$

Proposition 24 *Not* $a \sim_A b$.

Proof Let the context e be $-.x := \Omega^{\texttt{Bool}}.f$. Both $e[a]$ and $e[b]$ are programs of type `Bool`, but $e[a] \Downarrow$ `true` whereas $e[b]\Uparrow$. Hence the two are contextually distinct, therefore not bisimilar. ■

Proposition 25 $a \sim_{[x:\texttt{Bool}]} b$.

Proof Using (Eq Sub Object) we can prove both $[x = \texttt{true}] \sim_{[x:\texttt{Bool}]} a$ and $[x = \texttt{true}] \sim_{[x:\texttt{Bool}]} b$, and therefore $a \sim_{[x:\texttt{Bool}]} b$ by transitivity. ■

Proposition 26 $a \sim_{[f:\texttt{Bool}]} b$.

Proof Let P and Q be $a_{[f:\texttt{Bool}]}$ and $b_{[f:\texttt{Bool}]}$ respectively. Here are all their possible transitions.

(1) $P \xrightarrow{f} P'$ with $P' \equiv (a.f)_{\texttt{Bool}} \sim \texttt{true}_{\texttt{Bool}}$.

(2) $Q \xrightarrow{f} P'$ with $P' \equiv (b.f)_{\texttt{Bool}} \sim \texttt{true}_{\texttt{Bool}}$.

(3) $P \xrightarrow{f \Leftarrow \varsigma(x)e} P'$ with $P' \sim [x = \texttt{true}, f = \varsigma(x{:}A)e]_{[f:\texttt{Bool}]}$.

(4) $Q \xrightarrow{f \Leftarrow \varsigma(x)e} Q'$ with $Q' \sim [x = \texttt{true}, f = \varsigma(x{:}A)e]_{[f:\texttt{Bool}]}$.

In each case, whenever $P \xrightarrow{\alpha} P'$ there is Q' with $Q \xrightarrow{\alpha} Q'$ and $P' \sim Q'$, and vice versa. Hence $(P, Q) \in \langle\sim\rangle$ and since $\sim = \langle\sim\rangle$ we have $a \sim_{[f:\texttt{Bool}]} b$. ■

Proposition 26 does not follow from the equational theory $\leftrightarrow$. We expect it would follow by a direct proof of contextual equivalence (similar to the one we needed for (Eq Sub Object)) but the bisimulation proof above is much simpler.

8 Other equivalence relations

Our theory is based on characterising contextual equivalence as a form of bisimilarity. We considered several other forms of operational equivalence.

8.1 Contextual equivalence using capturing contexts

In Section 3, we defined contextual equivalence for closed expressions only. In extending the relation to open expressions, we have two choices; one is to use the relation $\simeq^\circ$, the other is to use contexts with a single hole that captures free variables; that is, we define a relation $\approx^1$ as follows.

$$e \approx^1 e' \quad \text{iff} \quad \begin{cases} \text{for all capturing contexts } \mathcal{C} \text{ s.t.} \\ \text{if } \mathcal{C}[e]{:}\texttt{Bool} \text{ and } \mathcal{C}[e']{:}\texttt{Bool} \text{ then} \\ \mathcal{C}[e] \Downarrow \text{ iff } \mathcal{C}[e'] \Downarrow \end{cases}$$

In the pure object calculus $\mathbf{Ob_{1<:\mu}}$ we can easily show that that $\approx^1$ contains contextual equivalence, but the reverse inclusion fails. The only bound variables in $\mathbf{Ob_{1<:\mu}}$ are self-parameters, of object type, so $x{:}\texttt{Bool} \vdash x \approx^1 \texttt{true}$ holds vacuously, but of course $x{:}\texttt{Bool} \vdash x \not\simeq^\circ \texttt{true}$, because $\texttt{false}/x$ is an $x{:}\texttt{Bool}$-closure. However, if the language is extended with a `let` construct at arbitrary type or with functions we can prove that the two equivalences are equal.

Proposition 27 *In the presence of functions or a* `let` *construct,* $\simeq^\circ = \approx^1$.

8.2 Record-style bisimilarity

Thinking of objects as records, we considered an equivalence, $\approx^2$, that equates two objects if selections of their methods are pairwise bisimilar.

$$a \approx^2_{[\ell_i:A_i](i\in I)} b \quad \text{iff} \quad \begin{cases} a \Downarrow \text{ iff } b \Downarrow \text{; and} \\ a.\ell \approx^2_{A_i} b.\ell \text{ for all } i \in I \end{cases}$$

This relation is not discriminatory enough because it has too narrow a notion of observation on objects. It would be correct for a record calculus, but it ignores the possibility of method update and indirection through self. For example, it equates a and b from Section 7 at type $[x{:}\texttt{Bool}, f{:}\texttt{Bool}]$, but we know from Proposition 24 that they are contextually distinct at that type. Abadi and Cardelli (1994c) reject a record-style semantics for their calculus for similar reasons.

8.3 Applicative bisimulation

Howe (1989) defines a format for applicative bisimulation, $\approx^3$, for a general class of untyped λ-calculi. Here is a natural way to express this format in a typed setting.

$$a \approx^3_A b \quad \text{iff} \quad \begin{array}{l} \text{whenever } a \Downarrow u \text{ then } \exists v \text{ s.t. } b \Downarrow v \\ \text{and } \varnothing \vdash u \mathrel{\widehat{\approx^{3\circ}}} v : A \text{, and vice versa.} \end{array}$$

Unfortunately, this format is too discriminatory for this object calculus. It distinguishes between the two programs a and b from Section 7 at the type $[f{:}\mathtt{Bool}]$, whereas Proposition 26 shows they are contextually equivalent. Two λ-calculus functions $\lambda(x{:}A)e$ and $\lambda(x{:}A)e'$ are equal if and only if e and e' are equal for any expression of the correct type that may be substituted for x. However, for the methods $\varsigma(s{:}A)\mathtt{true}$ and $\varsigma(s{:}A)s.x$ to be equal, their bodies only need to be equal when particular values of s are substituted for x, namely the objects a and b themselves.

9 Related work

Most prior work on the theoretical underpinnings for object-oriented programming uses denotational semantics (Gunter and Mitchell 1994), which provides fixpoint induction for reasoning about programs. Co-induction cannot always take the place of fixpoint induction, but Mason, Smith, and Talcott (1994) show how to derive fixpoint induction in a purely operational setting. Breazu-Tannen, Gunter, and Scedrov (1990) is one of the few papers to establish computational adequacy for a denotational semantics in the presence of subtyping. One conclusion of our study is that in spite of its elementary construction, bisimilarity is a useful operational model for an object calculus. Although much can be done purely operationally, it would be worthwhile to research the connections between contextual equivalence and the PER model for $\mathbf{Ob_{1<:\mu}}$.

Walker (1995) and Jones (1993) show how to encode objects in the π-calculus. Following their approach, we could translate $\mathbf{Ob_{1<:\mu}}$ into the π-calculus, but we expect, based on Sangiorgi (1994), that the equivalence generated by the encoding would be finer grained than contextual equivalence. Agha, Mason, Smith, and Talcott (1992) studied untyped actors, a form of objects, with side-effects and concurrency. We consider the extension of our results to the imperative object calculus of Abadi and Cardelli (1995) to be important future work. In the presence of dynamic state all known definitions of bisimilarity are finer grained than contextual equivalence (Stark 1994) but nonetheless we expect bisimilarity to be useful for imperative objects. $\mathbf{Ob_{1<:\mu}}$ is also studied by Palsberg (1994), who presents a complete type inference algorithm. $\mathbf{Ob_{1<:\mu}}$ is based on fixed-length objects; Mitchell, Honsell, and Fisher (1993) have developed a λ-calculus of extensible objects. They too define a simple operational semantics, analogous to our $\mapsto$ relation. We expect our theory of bisimilarity could be reworked for their calculus. We are aware of only two other studies of bisimilarity and subtyping. Pierce and Sangiorgi (1995) investigate type annotations on names in the π-calculus. Maung (1993), like us, used a labelled transition system and a notion of similarity to express object properties. He proved that similarity of his objects implies a notion of substitutability.

10 Conclusion

Contextual equivalence formally captures the idea that two programs are equal iff no amount of programming can tell them apart. We characterised contextual equivalence as a form of bisimilarity. We validated Abadi and Cardelli's equational theory. Furthermore, we showed that bisimilarity admits CCS-style proofs of equivalence, going beyond the equational theory. Our work builds on previous studies of bisimilarity for functional calculi (Abramsky and Ong 1993; Howe 1989; Crole and Gordon 1995; Gordon 1995). This is the first use of Howe's method in the presence of subsumption and the first study of contextual equivalence for an object calculus. The chief difficulties were in defining a labelled transition system that correctly dealt with method update and subsumption. Space precludes their inclusion here, but our main results extend to function, dynamic, record and variant types. In all we claim that operational methods are a promising new direction for the foundations of object-oriented programming.

Milner (1989) showed that bisimilarity is a useful theory of concurrent processes. Analogously, our work shows that bisimilarity is a useful theory of objects with subtyping. We have shown that from elementary foundations it captures intuitive operational arguments about objects.

Acknowledgements

Gordon holds a Royal Society University Research Fellowship. Rees holds an EPSRC Research Studentship. We thank Martín Abadi, Luca Cardelli and Andy Pitts for many useful conversations about this work.

References

Abadi, M. and L. Cardelli (1994a, June). A semantics of object types. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pp. 332–341. IEEE Computer Society Press.

Abadi, M. and L. Cardelli (1994b). A theory of primitive objects: Second-order systems. In *Proceedings of European Symposium on Programming*, Volume 788 of *Lecture Notes in Computer Science*, pp. 1–25. Springer-Verlag.

Abadi, M. and L. Cardelli (1994c, April). A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pp. 296–320. Springer-Verlag.

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

Abramsky, S. and L. Ong (1993). Full abstraction in the lazy lambda calculus. *Information and Computation 105*, 159–267. Available as Technical Report 259, University of Cambridge Computer Laboratory.

Agha, G., I. Mason, S. Smith, and C. Talcott (1992, August 24–27,). Towards a theory of actor computation. In *CONCUR'92: Third International Conference on Concurrency Theory, Stony Brook, New York*, Volume 630 of *Lecture Notes in Computer Science*, pp. 565–579. Springer-Verlag.

Breazu-Tannen, V., C. A. Gunter, and A. Scedrov (1990, June). Computing with coercions. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 44–60.

Crole, R. L. and A. D. Gordon (1995). A sound metalogical semantics for input/output effects. In *CSL'94 Computer Science Logic, Kazimierz, Poland, September 1994*, Volume 933 of *Lecture Notes in Computer Science*, pp. 339–353. Springer-Verlag.

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

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

Gordon, A. D. (1995). Bisimilarity as a theory of functional programming. In *Eleventh Annual Conference on Mathematical Foundations of Programming Semantics*, Volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B.V. To appear. Extended version available as BRICS Note NS–95–3, Aarhus University.

Gunter, C. A. and J. C. Mitchell (Eds.) (1994). *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, Cambridge, Mass.

Howe, D. J. (1989). Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pp. 198–203.

Jones, C. (1993). A pi-calculus semantics for an object-based design notation. In *CONCUR'93: Fourth International Conference on Concurrency Theory*, Volume 715 of *Lecture Notes in Computer Science*, pp. 158–172. Springer-Verlag.

Mason, I. A., S. F. Smith, and C. L. Talcott (1994). From operational semantics to domain theory. Submitted for publication.

Maung, I. (1993). Simulation, subtyping and substitutability. Technical Report UBC 93/5, Department of Computing, University of Brighton.

Meyer, A. R. and S. S. Cosmadakis (1988, July). Semantical paradigms: Notes for an invited lecture. In *Proceedings of the 3rd IEEE Sym-*

posium on Logic in Computer Science, pp. 236–253.

Milner, R. (1977). Fully abstract models of typed lambda-calculi. *Theoretical Computer Science 4*, 1–23.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall International.

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

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

Palsberg, J. (1994). Efficient inference of object types. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pp. 186–195.

Pierce, B. and D. Sangiorgi (1995). Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*. To appear. Summary in *Proceedings of the 8th IEEE Conference on Logic in Computer Science*, pp. 376–385 (1993).

Pitts, A. M. (1994). Computational adequacy via 'mixed' inductive definitions. In *Proceedings Mathematical Foundations of Programming Semantics IX, New Orleans 1993*, Volume 802 of *Lecture Notes in Computer Science*, pp. 72–82. Springer-Verlag.

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

Rees, G. (1994, April). Observational equivalence for a polymorphic lambda calculus. University of Cambridge Computer Laboratory. http://www.cl.cam.ac.uk/users/gdr11/equivalence.dvi.

Sangiorgi, D. (1994, May). The lazy lambda calculus in a concurrency scenario. *Information and Computation 111*(1), 120–153.

Stark, I. D. B. (1994, December). *Names and Higher-Order Functions*. Ph. D. thesis, University of Cambridge Computer Laboratory.

Walker, D. (1995, 1 February). Objects in the π-calculus. *Information and Computation 116*(2), 253–271.

$$\frac{}{\varnothing \vdash \diamond}\ (\text{Env } \varnothing) \qquad \frac{\Gamma \vdash \diamond \quad \varnothing \vdash A \quad x \notin Dom(\Gamma)}{\Gamma, x{:}A \vdash \diamond}\ (\text{Env } x) \qquad \frac{\Gamma \vdash E \quad X \notin Dom(\Gamma)}{\Gamma, X <: E \vdash \diamond}\ (\text{Env } X{<:})$$

Table 3: Well-formed environments

$$\frac{X \neq Y}{X \succ Y} \qquad \frac{}{\texttt{Top} \succ Y} \qquad \frac{}{\texttt{Bool} \succ Y} \qquad \frac{}{[\ell_i{:}E_i]_{i\in I} \succ Y} \qquad \frac{E \succ Y}{\mu(X)E \succ Y}$$

Table 4: Formal contractivity

$$\frac{\Gamma \vdash E_i\ (\forall i \in I)}{\Gamma \vdash [l_i : E_i]_{i\in I}}\ (\text{Type Object}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Top}}\ (\text{Type Top}) \qquad \frac{\Gamma, X <: E, \Gamma' \vdash \diamond}{\Gamma, X <: E, \Gamma' \vdash X}\ (\text{Type } X{<:})$$

$$\frac{\Gamma, X <: \texttt{Top} \vdash E \quad E \succ X}{\Gamma \vdash \mu(X)E}\ (\text{Type Rec } <:) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Bool}}\ (\text{Type Bool})$$

Table 5: Well-formed types

$$\frac{\Gamma \vdash E}{\Gamma \vdash E <: E}\ (\text{Sub Refl}) \qquad \frac{\Gamma \vdash E_1 <: E_2 \quad \Gamma \vdash E_2 <: E_3}{\Gamma \vdash E_1 <: E_3}\ (\text{Sub Trans}) \qquad \frac{\Gamma, X <: E, \Gamma' \vdash \diamond}{\Gamma, X <: E, \Gamma' \vdash X <: E}\ (\text{Sub } X)$$

$$\frac{\Gamma \vdash E}{\Gamma \vdash E <: \texttt{Top}}\ (\text{Sub Top}) \qquad \frac{J \subseteq I \quad \Gamma \vdash E_i\ (\forall i \in I)}{\Gamma \vdash [\ell_i : E_i]_{i\in I} <: [\ell_i : E_i]_{i\in J}}\ (\text{Sub Object})$$

$$\frac{\Gamma \vdash \mu(X_1)E_1 \quad \Gamma \vdash \mu(X_2)E_2 \quad \Gamma, X_2 <: \texttt{Top}, X_1 <: X_2 \vdash E_1 <: E_2}{\Gamma \vdash \mu(X_1)E_1 <: \mu(X_2)E_2}\ (\text{Sub Rec})$$

Table 6: Subtyping relation

$$\frac{\Gamma, x{:}A, \Gamma' \vdash \diamond}{\Gamma, x{:}A, \Gamma' \vdash x : A}\ (\text{Val } x) \qquad \frac{\Gamma, x_i{:}A \vdash e_i : A_i\ (\forall i \in I) \quad A \equiv [\ell_i{:}A_i]_{i\in I} \quad \Gamma \vdash \diamond}{\Gamma \vdash [\ell_i = \varsigma(x_i{:}A)e_i]_{i\in I} : A}\ (\text{Val Object})$$

$$\frac{\Gamma \vdash e : [\ell_i{:}A_i]_{i\in I} \quad j \in I}{\Gamma \vdash e.\ell_j : A_j}\ (\text{Val Select}) \qquad \frac{A \equiv [\ell_i{:}A_i]_{i\in I} \quad \Gamma \vdash e : A \quad \Gamma, x{:}A \vdash e' : A_j \quad j \in I}{\Gamma \vdash e.\ell_j \Leftarrow \varsigma(x{:}A)e' : A}\ (\text{Val Update})$$

$$\frac{A \equiv \mu(X)E \quad \Gamma \vdash e : E[A/X] \quad \varnothing \vdash A}{\Gamma \vdash \texttt{fold}(A, e) : A}\ (\text{Val Fold}) \qquad \frac{A \equiv \mu(X)E \quad \Gamma \vdash e : A}{\Gamma \vdash \texttt{unfold}(e) : E[A/X]}\ (\text{Val Unfold})$$

$$\frac{\Gamma \vdash e : A_1 \quad \Gamma \vdash A_1 <: A_2}{\Gamma \vdash e : A_2}\ (\text{Val Subsumption}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{true} : \texttt{Bool}}\ (\text{Val True})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{false} : \texttt{Bool}}\ (\text{Val False}) \qquad \frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e_3 : A}{\Gamma \vdash \texttt{if}(e_1, e_2, e_3) : A}\ (\text{Val If})$$

Table 7: Type assignment