A logic of object-oriented programs

Martín Abadi and K. Rustan M. Leino

Systems Research Center
Digital Equipment Corporation
{ma,rustan}@pa.dec.com

Abstract. We develop a logic for reasoning about object-oriented programs. The logic is for a language with an imperative semantics and aliasing, and accounts for self-reference in objects. It is much like a type system for objects with subtyping, but our specifications go further than types in detailing pre- and postconditions. We intend the logic as an analogue of Hoare logic for object-oriented programs. Our main technical result is a soundness theorem that relates the logic to a standard operational semantics.

1 Introduction

In the realm of procedural programming, Floyd and Hoare defined two of the first logics of programs [Flo67, Hoa69]; many later formalisms and systems built on their ideas, and addressed difficult questions of concurrency and data abstraction, for example. An analogous development has not taken place in object-oriented programming. Although there is much formal work on objects (see section 6), the literature on objects does not seem to contain an analogue for Floyd's logic or Hoare's logic. In our opinion, this is an important gap in the understanding of object-oriented programming languages.

Roughly imitating Hoare, we develop a logic for the specification and verification of object-oriented programs. We focus on elementary goals: we are interested in logical reasoning about pre- and postconditions of programs written in a basic object-oriented programming language (a variant of the calculi of Abadi and Cardelli [AC96]). Like Hoare, we deal with partial correctness, not with termination.

The programming language presents many interesting and challenging features of common object-oriented languages. In particular, the operational semantics of the language is imperative and allows aliasing. Objects have fields and methods, and the self variable permits self-reference. At the type level, the type of an object lists the types of its fields and the result types of its methods; a subtyping relation supports subsumption and inheritance. We mostly ignore "advanced" issues, like concurrency, but some of them have been considered in the literature (e.g., see [Jon92, YT87]).

Much like Hoare logic, our logic includes one rule for reasoning about pre- and postconditions for each of the constructs of the programming language. In order to formulate these rules, we introduce object specifications. An object specification is a generalization of an object type: it lists the specifications of fields, the specifications of the methods' results, and also gives the pre/postcondition descriptions of the methods.

Some of the main advantages of Hoare logic are its formal precision and its simplicity. These advantages make it possible to study Hoare logic, and for example to prove its soundness and completeness; they also make it easier to extend and to implement Hoare logic. We aim to develop a logic with some of those same advantages. Our rules are not quite as simple as Hoare's, in part because of aliasing, and in part because objects are more expressive than first-order procedures and give some facilities for higher-order programming (cf. [Cla79, Apt81]). However, our rules are precise; in particular, we are able to state and to prove a soundness theorem. We do not know of any equivalent soundness theorem in the object-oriented literature.

In the next section we describe the programming language. In section 3 we develop a logic for this language, and in section 4 we give some examples of the use of this logic in verification. In section 5, we discuss soundness and completeness with respect to the operational semantics of section 2. Finally, in sections 6 and 7, we review some related work, discuss possible extensions of our own work, and conclude.

2 The language

In this section we define a small object-oriented language similar to the calculi of Abadi and Cardelli. Those calculi have few syntactic forms, but are quite expressive. They are object-based; they do not include primitives for classes and inheritance, which can be simulated using simpler constructs.

We give the syntax of our language, its operational semantics, and a set of type rules. These aspects of the language are (intentionally) not particularly novel or exotic; we describe them only as background for the rest of the paper.

2.1 Syntax and operational semantics

We assume we are given a set $\mathcal{V}$ of program variables (written x, y, z, and w possibly with subscripts), a set $\mathcal{F}$ of field names (written f and g, possibly with subscripts), and a set $\mathcal{M}$ of method names (written m, possibly with subscripts). These sets are disjoint. The grammar of the language is:

a, b	$::=$	x	variables
	$\mid$	$\mathit{false} \mid \mathit{true}$	constants
	$\mid$	$\mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1$	conditional
	$\mid$	$\mathit{let}\ x = a\ \mathit{in}\ b$	let
	$\mid$	$[\mathrm{f}_i = x_i\ ^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j)b_j\ ^{j \in 1..m}]$	object construction
	$\mid$	$x.\mathrm{f}$	field selection
	$\mid$	$x.\mathrm{m}$	method invocation
	$\mid$	$x.\mathrm{f} := y$	field update

Throughout, we assume that the names f_i and m_j are all distinct in the construct $[\mathrm{f}_i = x_i \ ^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j) b_j \ ^{j \in 1..m}]$, and we allow the renaming of bound variables in all expressions.

Informally, the semantics of the language is as follows:

- Variables are identifiers; they are not mutable: $x := a$ is not a legal statement. This restriction is convenient but not fundamental. (We can simulate assignment by binding a variable to an object with a single field and updating that field.)
- *false* and *true* evaluate to themselves.
- *if* x *then* a_0 *else* a_1 evaluates a_0 if x is *true* and evaluates a_1 if x is *false*.
- *let* $x = a$ *in* b evaluates a and then evaluates b with x bound to the result of a. We define $a \ ; b$ as a shorthand for *let* $x = a$ *in* b where x does not occur free in b.
- $[\mathrm{f}_i = x_i \ ^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j) b_j \ ^{j \in 1..m}]$ creates and returns a new object with fields f_i and methods m_j. The initial value for the field f_i is the value of x_i. The method m_j is set to $\varsigma(y_j) b_j$, where ς is a binder, y_j is a variable (the self parameter of the method), and b_j is a program (the body of the method).
- Fields can be both selected and updated. In the case of selection ($x.\mathrm{f}$), the value of the field is returned; in the case of update ($x.\mathrm{f} := y$), the value of the object is returned.
- When a method of an object is invoked ($x.\mathrm{m}$), its self variable is bound to the object itself and the body of the method is executed. The method does not have any explicit parameters besides the self variable; however, additional parameters can be passed via the fields of the object.

Objects are references (rather than records), and the semantics allows aliasing. For example, the program fragment

$$let\ x = [\mathrm{f} = z_0]\ in\ let\ y = x\ in\ (x.\mathrm{f} := z_1\ ;\ y.\mathrm{f})$$

allocates some storage, creates two references to it (x and y), updates the storage through x, and then reads it through y, returning z_1.

The semantics can be defined more formally in terms of stacks and stores. A stack maps variables to values (booleans or references). A store contains values for object fields and closures for object methods. We write $\sigma, S \vdash b \rightsquigarrow v, \sigma'$ to mean that, given initial store σ and stack S, executing the program b leads to the result v and to the final store σ'. (We leave details to an extended version of this paper.)

We have defined a small language in order to simplify the presentation of our rules. In examples, we sometimes extend the syntax with additional, standard constructs, such as integers. The rules for such constructs are straightforward.

2.2 Types

We present a first-order type system for our language. The types are *Bool* and object types, which have the form $[\mathrm{f}_i \colon A_i \ ^{i \in 1..n},\ \mathrm{m}_j \colon B_j \ ^{j \in 1..m}]$. This is the type of objects with a field f_i of type A_i, for $i \in 1..n$, and with a method m_j with result type B_j, for $j \in 1..m$. The order of the components does not matter.

The type system includes a reflexive and transitive subtyping relation. A longer object type is a subtype of a shorter one, and in addition object types are covariant in the result types of methods. More precisely, the type $[f_i{:}\, A_i\ ^{i \in 1..n+p}, m_j{:}\, B_j\ ^{j \in 1..m+q}]$ is a subtype of $[f_i{:}\, A_i\ ^{i \in 1..n}, m_j{:}\, B'_j\ ^{j \in 1..m}]$ provided B_j is a subtype of B'_j, for $j \in 1..m$. Thus, object types are invariant in the types of fields; this invariance is essential for soundness [AC96].

Formally, we write $\vdash A$ to express that A is a well-formed type, and $\vdash A <: A'$ to express that A is a subtype of A'. We have the rules:

Well-formed types

$$\frac{}{\vdash \mathit{Bool}} \qquad \frac{\vdash A_i\ ^{i \in 1..n} \qquad \vdash B_j\ ^{j \in 1..m}}{\vdash [f_i{:}\, A_i\ ^{i \in 1..n},\ m_j{:}\, B_j\ ^{j \in 1..m}]}$$

Subtypes

$$\frac{}{\vdash \mathit{Bool} <: \mathit{Bool}}$$

$$\frac{\vdash A_i\ ^{i \in 1..n+p} \qquad \vdash B_j <: B'_j\ ^{j \in 1..m} \qquad \vdash B_j\ ^{j \in m+1..m+q}}{\vdash [f_i{:}\, A_i\ ^{i \in 1..n+p},\ m_j{:}\, B_j\ ^{j \in 1..m+q}] <: [f_i{:}\, A_i\ ^{i \in 1..n},\ m_j{:}\, B'_j\ ^{j \in 1..m}]}$$

A typing environment is a (possibly empty) list of pairs $x{:}\, A$, where x is a variable and A is a type. The variables of each environment are distinct. We write $\emptyset$ for the empty environment, and say that x is in E when it appears in some pair $x{:}\, A$ in E. We write $E \vdash \diamond$ to express that E is a well-formed typing environment. We have two rules for forming typing environments:

Well-formed typing environments

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{E \vdash \diamond \qquad \vdash A \qquad x \text{ not in } E}{E, x{:}\, A \vdash \diamond}$$

We write $E \vdash a : A$ to express that, in environment E, program a has type A. There is one typing rule for each construct, and an additional rule for subsumption. We write $\stackrel{\scriptscriptstyle syn}{=}$ for the relation of syntactic equality (up to reordering of object components).

Well-typed programs

Subsumption

$$\frac{\vdash A <: A' \qquad E \vdash a : A}{E \vdash a : A'}$$

Variables

$$\frac{E, x{:}\, A, E' \vdash \diamond}{E, x{:}\, A, E' \vdash x : A}$$

Constants

$$\frac{E \vdash \diamond}{E \vdash \mathit{false} : \mathit{Bool}} \qquad \frac{E \vdash \diamond}{E \vdash \mathit{true} : \mathit{Bool}}$$

Conditional

$$\frac{E \vdash x : Bool \qquad E \vdash a_0 : A \qquad E \vdash a_1 : A}{E \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 : A}$$

Let

$$\frac{E \vdash a : A \qquad E, x{:}\, A \vdash b : B}{E \vdash \mathit{let}\ x = a\ \mathit{in}\ b : B}$$

Object construction for $A \stackrel{\text{syn}}{=} [\mathrm{f}_i{:}\, A_i\ {}^{i \in 1..n},\ \mathrm{m}_j{:}\, B_j\ {}^{j \in 1..m}]$

$$\frac{E \vdash \diamond \qquad E \vdash x_i : A_i\ {}^{i \in 1..n} \qquad E, y_j{:}\, A \vdash b_j : B_j\ {}^{j \in 1..m}}{E \vdash [\mathrm{f}_i = x_i\ {}^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j) b_j\ {}^{j \in 1..m}] : A}$$

Field selection

$$\frac{E \vdash x : [\mathrm{f}{:}\, A]}{E \vdash x.\mathrm{f} : A}$$

Method invocation

$$\frac{E \vdash x : [\mathrm{m}{:}\, B]}{E \vdash x.\mathrm{m} : B}$$

Field update for $A \stackrel{\text{syn}}{=} [\mathrm{f}_i{:}\, A_i\ {}^{i \in 1..n},\ \mathrm{m}_j{:}\, B_j\ {}^{j \in 1..m}]$

$$\frac{E \vdash x : A \qquad k \in 1..n \qquad E \vdash y : A_k}{E \vdash x.\mathrm{f}_k := y : A}$$

This type system is like those of common programming languages in that it is independent of verification rules. In particular, types are not automatically associated with specifications, and subtyping does not impose any "behavioral" constraints (cf. [LW94]). However, as section 3 explains, specifications generalize types.

3 Verification

In this section, which is the core of the paper, we give rules for verifying object-oriented programs written in the language of section 2. We start with an informal explanation of our approach.

3.1 Transition relations

The purpose of our verification rules is to allow reasoning about pre- and post-conditions. These pre- and postconditions concern the initial and final stores, the stack, and the result of the execution of a given program.

In our rules, we express pre- and postconditions in formulas of standard, untyped first-order logic that we call *transition relations*. These formulas mention the unary predicates $\grave{alloc}$ and $\acute{alloc}$, two binary functions $\grave{\sigma}$ and $\acute{\sigma}$, and the special variable r (which is not in the set $\mathcal{V}$ of program variables). Intuitively, $\grave{\sigma}(x, \mathrm{f})$ is the value of field f of object x before the execution, and $\acute{\sigma}(x, \mathrm{f})$ is its value after the execution. Similarly, $\mathit{al\grave{l}oc}(x)$ and $\mathit{al\acute{l}oc}(x)$ indicate whether x has been allocated before and after the execution. Finally, the variable r represents the result of the execution.

For example, we may want to prove that, after any execution of the program $x.\mathrm{f} := y$, the result is x and the field f of x equals y. We can express this with the transition relation $r = x \wedge \acute{\sigma}(x, \mathrm{f}) = y$. As a second example, we may want to prove that, after any execution of $x.\mathrm{f}$, the result equals the initial value of the field f of x, and that the store is not changed by the execution. This statement is captured by the transition relation $r = \grave{\sigma}(x, \mathrm{f}) \wedge (\forall y, z \,.\, \grave{\sigma}(y, z) = \acute{\sigma}(y, z) \wedge (\mathit{al\grave{l}oc}(y) \equiv \mathit{al\acute{l}oc}(y))\ )$.

We work in standard first-order logic, so the functions $\grave{\sigma}$ and $\acute{\sigma}$ are total. Hence, $\grave{\sigma}(x, \mathrm{f})$ and $\acute{\sigma}(x, \mathrm{f})$ are defined even if $\mathit{al\grave{l}oc}(x)$ and $\mathit{al\acute{l}oc}(x)$ do not hold. In that case, the values of $\grave{\sigma}(x, \mathrm{f})$ and $\acute{\sigma}(x, \mathrm{f})$ are not important.

Given a program, a transition relation is much like a Hoare triple from the point of view of expressiveness. For example, a transition relation such as $(\grave{\sigma}(x, \mathrm{f}) = \grave{\sigma}(x, \mathrm{g})) \Rightarrow (\acute{\sigma}(x, \mathrm{f}) = \acute{\sigma}(x, \mathrm{g}))$ can be understood as assuming a precondition $(\grave{\sigma}(x, \mathrm{f}) = \grave{\sigma}(x, \mathrm{g}))$ and asserting a postcondition $(\acute{\sigma}(x, \mathrm{f}) = \acute{\sigma}(x, \mathrm{g}))$. However, the precondition and postcondition are given by separate formulas in a Hoare triple, while there is no such formal separation in a transition relation. This difference is largely a matter of convenience.

Formally, we write that T is a transition relation to mean that T is a well-formed formula of the standard, untyped first-order logic, made up only of:

- the constants *false* and *true*;
- the variable r, the binary functions $\grave{\sigma}$ and $\acute{\sigma}$, and the unary predicates $\mathit{al\grave{l}oc}$ and $\mathit{al\acute{l}oc}$;
- constants for field names (such as f);
- other variables (such as x);
- the usual logical connectives $\neg$, $\wedge$, and $\forall$ (from which $\vee$, $\Rightarrow$, $\equiv$, and $\exists$ can be defined as abbreviations), and the equality predicate $=$.

The grammar for transition relations is thus:

$$\begin{array}{lcl} T & ::= & e_0 = e_1 \mid \mathit{al\grave{l}oc}(e) \mid \mathit{al\acute{l}oc}(e) \mid \neg T \mid T_0 \wedge T_1 \mid (\forall x \,.\, T\ ) \\ e & ::= & \mathit{false} \mid \mathit{true} \mid r \mid x \mid \mathrm{f} \mid \grave{\sigma}(e_0, e_1) \mid \acute{\sigma}(e_0, e_1) \end{array}$$

3.2 Specifications and subspecifications

In order to permit reasoning about pre- and postconditions, our verification rules also deal with *specifications*, which generalize types. A specification can be either *Bool* or an *object specification*, of the form:

$$[\mathrm{f}_i \colon A_i \ {}^{i \in 1..n},\ \mathrm{m}_j \colon \varsigma(y_j) B_j :: T_j \ {}^{j \in 1..m}]$$

where each A_i and B_j is a specification, and each T_j is a transition relation. The variable y_j is bound in B_j and T_j. An object satisfies the specification $[\mathrm{f}_i \colon A_i \ {}^{i \in 1..n},\ \mathrm{m}_j \colon \varsigma(y_j) B_j :: T_j \ {}^{j \in 1..m}]$ if, for $i \in 1..n$, it has a field f_i that satisfies specification A_i, and, for $j \in 1..m$, it has a method m_j with a result that satisfies B_j and whose execution satisfies T_j when y_j equals self. Informally, we may think of B_j as a predicate on the result, and then we may read $B_j :: T_j$ as the conjunction of that predicate and T_j. As for object types, the order of the components of object specifications does not matter.

Just like there is a subtyping relation on types, there is a *subspecification* relation on specifications. This relation is reflexive and transitive. A longer object specification is a subspecification of a shorter one, and in addition object specifications are covariant in the result specifications and in the transition relations for methods. Intuitively, when A and A' are object specifications, A is a subspecification of A' only if any object that satisfies A also satisfies A'.

3.3 Rules for specifications

In our rules for specifications, we use several judgments analogous to those introduced for types in section 2.2, and in those cases we use similar notations but with a $\Vdash$ instead of a $\vdash$. In particular, we write $\Vdash A$ to express that A is a well-formed specification, and $\Vdash A <: A'$ to express that A is a subspecification of A'. The following rules for specifications generalize the corresponding rules for types:

Well-formed specifications

$$\frac{}{\Vdash \mathit{Bool}} \qquad \frac{\Vdash A_i \;{}^{i \in 1..n} \qquad \Vdash B_j \;{}^{j \in 1..m} \qquad T_j \text{ is a transition relation } {}^{j \in 1..m}}{\Vdash [\mathrm{f}_i : A_i \;{}^{i \in 1..n},\ \mathrm{m}_j : \varsigma(y_j) B_j :: T_j \;{}^{j \in 1..m}]}$$

Subspecifications

$$\frac{}{\Vdash \mathit{Bool} <: \mathit{Bool}}$$

$$\frac{\begin{array}{l}\Vdash A_i \;{}^{i \in 1..n+p} \qquad \Vdash B_j <: B'_j \;{}^{j \in 1..m} \qquad \Vdash B_j \;{}^{j \in m+1..m+q} \\ \Vdash_{fol} T_j \Rightarrow T'_j \;{}^{j \in 1..m} \\ T_j \text{ is a transition relation } {}^{j \in 1..m+q} \qquad T'_j \text{ is a transition relation } {}^{j \in 1..m}\end{array}}{\begin{array}{c}\Vdash [\mathrm{f}_i : A_i \;{}^{i \in 1..n+p},\ \mathrm{m}_j : \varsigma(y_j) B_j :: T_j \;{}^{j \in 1..m+q}] \\ <: [\mathrm{f}_i : A_i \;{}^{i \in 1..n},\ \mathrm{m}_j : \varsigma(y_j) B'_j :: T'_j \;{}^{j \in 1..m}]\end{array}}$$

In this last rule, $\Vdash_{fol}$ represents provability in first-order logic.

3.4 Specification environments

A *specification environment* is much like a typing environment, except that it contains specifications instead of types. We write $E \Vdash \diamond$ to mean that E is a well-formed specification environment. We have the rules:

Well-formed specification environments

$$\frac{}{\emptyset \Vdash \diamond} \qquad \frac{E \Vdash \diamond \qquad E \Vdash A \qquad x \text{ not in } E}{E, x : A \Vdash \diamond}$$

Here, given a well-formed specification environment E, we write $E \Vdash A$ to mean $\Vdash A$ and that all the free program variables of A are in E. We omit the obvious rule for this judgment. Similarly, when all the free program variables of a transition relation T are in E, we write:

$$E \Vdash T \text{ is a transition relation}$$

In order to formulate the verification rules, we introduce the judgment:

$$E \Vdash a : A :: T$$

This judgment states that, in specification environment E, the execution of a satisfies the transition relation T, and its result satisfies the specification A.

For this judgment, there is one rule per construct plus a subsumption rule; the rules are all given below. The rules guarantee that, whenever $E \Vdash a : A :: T$ is provable, all the free program variables of a, A, and T are in E. The rules have interesting similarities both with the operational semantics and with the typing rules. The treatment of transition relations reiterates parts of the operational semantics, while the treatment of specifications generalizes that of types.

The subsumption rule enables us to weaken a specification and a transition relation, much like we weaken a type in the subsumption rule for typing. The rule for *if-then-else* allows the replacement of the boolean guard with its value in reasoning about each of the alternatives. The rule for *let* achieves sequencing by representing an intermediate state with the auxiliary binary function $\check{\sigma}$ and unary predicate $al\check{l}oc$. The variable x bound by *let* cannot escape because of the hypotheses that $E \Vdash B$ and that $E \Vdash T''$ is a transition relation. The rule for object construction has a complicated transition relation, but this transition relation directly reflects the operational semantics; the introduction of an object specification requires the verification of the methods of the new object. The rule for method invocation takes advantage of an object specification for yielding a specification and a transition relation; in these, the formal self is replaced with the actual self. The remaining rules are mostly straightforward.

In several rules, we use transition relations of the form $Res(e)$, where e is a term; $Res(e)$ is defined by:

$$Res(e) \quad \triangleq \quad r = e \;\wedge\; (\forall\, x, y \,.\, \grave{\sigma}(x, y) = \acute{\sigma}(x, y) \;\wedge\; (al\grave{l}oc(x) \;\equiv\; al\acute{l}oc(x))\,)$$

and it means that the result is e and that the store does not change. We also write $u_1[u_2/u_3]$ for the result of substituting u_2 for u_3 in u_1.

Well-specified programs

Subsumption

$$\frac{\begin{array}{lll} \Vdash A <: A' & \Vdash_{fol} T \Rightarrow T' & E \Vdash a : A :: T \\ E \Vdash A' & \multicolumn{2}{l}{E \Vdash T' \text{ is a transition relation}} \end{array}}{E \Vdash a : A' :: T'}$$

Variables

$$\frac{E, x{:}\, A, E' \Vdash \diamond}{E, x{:}\, A, E' \Vdash x : A :: Res(x)}$$

Constants

$$\frac{E \Vdash \diamond}{E \Vdash \mathit{false} : \mathit{Bool} :: \mathit{Res}(\mathit{false})} \qquad \frac{E \Vdash \diamond}{E \Vdash \mathit{true} : \mathit{Bool} :: \mathit{Res}(\mathit{true})}$$

Conditional

$$\frac{\begin{array}{l} E \Vdash x : \mathit{Bool} :: \mathit{Res}(x) \\ E \Vdash a_0 : A_0 :: T_0 \quad A_0[\mathit{true}/x] \overset{\scriptscriptstyle syn}{=} A[\mathit{true}/x] \quad T_0[\mathit{true}/x] \overset{\scriptscriptstyle syn}{=} T[\mathit{true}/x] \\ E \Vdash a_1 : A_1 :: T_1 \quad A_1[\mathit{false}/x] \overset{\scriptscriptstyle syn}{=} A[\mathit{false}/x] \quad T_1[\mathit{false}/x] \overset{\scriptscriptstyle syn}{=} T[\mathit{false}/x] \end{array}}{E \Vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 : A :: T}$$

Let

$$\frac{\begin{array}{l} E \Vdash a : A :: T \qquad E, x{:}A \Vdash b : B :: T' \\ E \Vdash B \qquad E \Vdash T'' \text{ is a transition relation} \\ \Vdash_{fol} T[\check{\sigma}/\acute{\sigma}, \check{alloc}/\acute{alloc}, x/r] \wedge T'[\check{\sigma}/\grave{\sigma}, \check{alloc}/\grave{alloc}] \Rightarrow T'' \end{array}}{E \Vdash \mathit{let}\ x = a\ \mathit{in}\ b : B :: T''}$$

Object construction for $A \overset{\scriptscriptstyle syn}{=} [\mathrm{f}_i{:}\, A_i\ ^{i \in 1..n},\ \mathrm{m}_j{:}\, \varsigma(y_j) B_j :: T_j\ ^{j \in 1..m}]$

$$\frac{E \Vdash \diamond \qquad E \Vdash x_i : A_i :: \mathit{Res}(x_i)\ ^{i \in 1..n} \qquad E, y_j{:}\, A \Vdash b_j : B_j :: T_j\ ^{j \in 1..m}}{\begin{array}{l} E \Vdash [\mathrm{f}_i = x_i\ ^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j) b_j\ ^{j \in 1..m}] : A :: \\ \qquad \neg \grave{alloc}(r) \wedge \acute{alloc}(r) \wedge \\ \qquad (\forall z \,.\, z \neq r \Rightarrow (\grave{alloc}(z) \equiv \acute{alloc}(z))\ ) \wedge \\ \qquad \acute{\sigma}(r, \mathrm{f}_1) = x_1 \wedge \cdots \wedge \acute{\sigma}(r, \mathrm{f}_n) = x_n \wedge \\ \qquad (\forall z, w \,.\, z \neq r \Rightarrow \grave{\sigma}(z, w) = \acute{\sigma}(z, w)\ ) \end{array}}$$

Field selection

$$\frac{E \Vdash x : [\mathrm{f}{:}\, A] :: \mathit{Res}(x)}{E \Vdash x.\mathrm{f} : A :: \mathit{Res}(\grave{\sigma}(x, \mathrm{f}))}$$

Method invocation

$$\frac{E \Vdash x : [\mathrm{m}{:}\, \varsigma(y) B :: T] :: \mathit{Res}(x)}{E \Vdash x.\mathrm{m} : B[x/y] :: T[x/y]}$$

Field update for $A \overset{\scriptscriptstyle syn}{=} [\mathrm{f}_i{:}\, A_i\ ^{i \in 1..n},\ \mathrm{m}_j{:}\, \varsigma(z_j) B_j :: T_j\ ^{j \in 1..m}]$

$$\frac{E \Vdash x : A :: \mathit{Res}(x) \qquad k \in 1..n \qquad E \Vdash y : A_k :: \mathit{Res}(y)}{\begin{array}{l} E \Vdash x.\mathrm{f}_k := y : A :: \\ \qquad r = x \wedge \acute{\sigma}(x, \mathrm{f}_k) = y \wedge \\ \qquad (\forall z, w \,.\, \neg(z = x \wedge w = \mathrm{f}_k) \Rightarrow \grave{\sigma}(z, w) = \acute{\sigma}(z, w)\ ) \wedge \\ \qquad (\forall z \,.\, \grave{alloc}(z) \equiv \acute{alloc}(z)\ ) \end{array}}$$

4 Examples

We discuss a few instructive examples (omitting derivations for brevity). From now on, we use some abbreviations, allowing general expressions to appear where the grammar requires a variable. For a, $a_i\ ^{i \in 1..n}$, and b not variables, we define:

$$
\begin{array}{rcl}
\textit{if } b \textit{ then } a_0 \textit{ else } a_1 & \triangleq & \textit{let } x = b \textit{ in if } x \textit{ then } a_0 \textit{ else } a_1 \\
[\mathrm{f}_i = a_i \ ^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j)b_j \ ^{j \in 1..m}] & \triangleq & \textit{let } x_1 = a_1 \textit{ in } \cdots \textit{ let } x_n = a_n \textit{ in} \\
 & & [\mathrm{f}_i = x_i \ ^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j)b_j \ ^{j \in 1..m}] \\
a.\mathrm{f} & \triangleq & \textit{let } x = a \textit{ in } x.\mathrm{f} \\
a.\mathrm{m} & \triangleq & \textit{let } x = a \textit{ in } x.\mathrm{m} \\
a.\mathrm{f} := b & \triangleq & \textit{let } x = a \textit{ in} \\
 & & (x.\mathrm{f}\ ;\ \textit{let } y = b \textit{ in } x.\mathrm{f} := y)
\end{array}
$$

where the variables x and $x_i\ ^{i \in 1..n}$ are fresh. Rules for these abbreviations can be derived directly from the rules for the language proper.

Field update and selection Our first example concerns the program:

$$([\mathrm{f} = \textit{false}].\mathrm{f} := \textit{true}).\mathrm{f}$$

This program constructs an object with one field, f, whose initial value is *false*. It then updates the value of the field to *true*. Finally, a field selection retrieves the new value of the field.

Using our rules, we can prove that $r = \textit{true}$ holds upon termination of this program. Formally, we can derive the judgment:

$$\emptyset \Vdash ([\mathrm{f} = \textit{false}].\mathrm{f} := \textit{true}).\mathrm{f} : \textit{Bool} :: (r = \textit{true})$$

Aliasing The following three programs exhibit the rôle of aliasing:

$$
\begin{array}{l}
\textit{let } x = [\mathrm{f} = \textit{false}] \textit{ in let } y = [\mathrm{g} = \textit{false}] \textit{ in } (y.\mathrm{g} := \textit{true}\ ;\ x.\mathrm{f}) \\
\textit{let } x = [\mathrm{f} = \textit{false}] \textit{ in let } y = [\mathrm{f} = \textit{false}] \textit{ in } (y.\mathrm{f} := \textit{true}\ ;\ x.\mathrm{f}) \\
\textit{let } x = [\mathrm{f} = \textit{true}] \textit{ in let } y = x \textit{ in } (y.\mathrm{f} := \textit{false}\ ;\ x.\mathrm{f})
\end{array}
$$

For each of these programs we can verify that $r = \textit{false}$. The first program shows that an update of a field g has no effect on another field f. The second program shows that separately constructed objects have different fields, even if those fields have the same name. The third program shows that an update of a field of an aliased object can be seen through all the aliases.

Method invocations and recursion The next example illustrates the use of method invocation; it shows how object specifications play the rôle of loop invariants for recursive method invocations.

We consider an object-oriented implementation of Euclid's algorithm for computing greatest common divisors. This implementation uses an object with two fields, f and g, and a method m:

$$
\begin{array}{l}
[\ \mathrm{f} = 1,\ \mathrm{g} = 1, \\
\ \ \mathrm{m} = \varsigma(y)\ \textit{if } y.\mathrm{f} < y.\mathrm{g} \textit{ then } (y.\mathrm{g} := y.\mathrm{g} - y.\mathrm{f}\ ;\ y.\mathrm{m}) \\
\qquad\qquad\quad \textit{else if } y.\mathrm{g} < y.\mathrm{f} \textit{ then } (y.\mathrm{f} := y.\mathrm{f} - y.\mathrm{g}\ ;\ y.\mathrm{m}) \\
\qquad\qquad\quad \textit{else } y.\mathrm{f}\ ]
\end{array}
$$

Setting f and g to two positive integer values and then invoking the method m has the effect of reducing both f and g to the greatest common divisor of those two values.

We can prove that this object satisfies the following specification:

$$
\begin{aligned}
&[\ \mathrm{f}{:}\, Nat,\ \ \mathrm{g}{:}\, Nat,\\
&\quad \mathrm{m}{:}\, \varsigma(y)\ Nat\ ::\ 1 \leq \grave{\sigma}(y,\mathrm{f}) \wedge 1 \leq \grave{\sigma}(y,\mathrm{g}) \ \ \Rightarrow\\
&\qquad\qquad\qquad r = \acute{\sigma}(y,\mathrm{f}) \wedge r = \acute{\sigma}(y,\mathrm{g}) \wedge r = gcd(\grave{\sigma}(y,\mathrm{f}), \grave{\sigma}(y,\mathrm{g}))\ ]
\end{aligned}
$$

In verifying the body of m, we can use the specification of m, recursively.

Nontermination As we mentioned initially, our rules are for partial correctness, not for termination. Nontermination can easily arise because of recursive method invocations. Consider, for example, the nonterminating program:

$$[\mathrm{m} = \varsigma(x)\, x.\mathrm{m}].\mathrm{m}$$

Using our rules, we can prove that anything holds upon termination of this program, vacuously. Formally, we can derive the judgment:

$$\emptyset \Vdash [\mathrm{m} = \varsigma(x)\, x.\mathrm{m}].\mathrm{m} : A :: T$$

for any closed specification A and transition relation T.

5 Soundness and related properties

In this section we discuss the relation between verification and typing, obtaining two simple results. We then discuss the relation between verification and operational semantics, proving in particular a soundness theorem. The soundness theorem is the main technical result of this paper. Finally, we comment on completeness.

5.1 Typing versus verification

Our first result establishes a correspondence between typing rules and verification rules: it says that only well-typed programs can be verified.

Proposition 1. *If $E \Vdash a : A :: T$ then $E' \vdash a : A'$ for some E' and A' (obtained from E and A by deleting transition relations).*

This result provides a first sanity check for the verification rules. It also highlights a limitation: for example, it implies that the verification rules do not enable us to derive that the program *if true then true else* (*true*.f) yields $r = true$, because this program is not well-typed. We do not view this limitation as a serious one because we are primarily interested in well-typed programs.

Conversely, all well-typed programs can be verified, at least in a trivial sense:

Proposition 2. *If $E' \vdash a : A'$ then $E \Vdash a : A :: (r = r)$ for some E and A (obtained from E' and A' by inserting trivial transition relations).*

5.2 Soundness

We have both an axiomatic semantics (the verification rules) and an operational semantics. Fortunately, the two semantics agree in the sense that all that can be derived with the verification rules is true operationally. For example, if a program yields a result according to the operational semantics, and the axiomatic semantics says that the result is *true*, then indeed the result is *true*. This property is expressed by the following soundness theorem:

Theorem 3. *Assume that the operational semantics says that program b yields result v when run with an empty stack and an empty initial store (that is, $\emptyset, \emptyset \vdash b \rightsquigarrow v, \sigma'$ for some σ'). If $\emptyset \Vdash b : \mathit{Bool} :: (r = \mathit{true})$ is provable then v is the boolean true. Similarly, if $\emptyset \Vdash b : \mathit{Bool} :: (r = \mathit{false})$ is provable then v is the boolean false.*

In an extended version of this work, we prove a more general soundness theorem in full. Theorem 3 is a corollary of that more general theorem. As another corollary, we obtain a soundness theorem for the type system of section 2.2. Therefore, as might be expected, our proofs are no less intricate than typical soundness proofs for type systems of imperative languages. In fact, they generalize techniques developed for proofs of type soundness [Har94, Ler92, Tof90, WF94]. New ingredients are required because specifications, unlike ordinary (non-dependent) types, may contain occurrences of program variables.

5.3 Completeness issues

While we have soundness, we do not have its converse, completeness. Unfortunately, our rules do not seem to be complete even for well-typed programs.

Careful examination of the following three similar programs reveals a first difficulty:

$$\begin{array}{lcl} b_1 & \triangleq & \mathit{let}\ x = (\mathit{let}\ y = \mathit{true}\ \mathit{in}\ [\mathrm{m} = \varsigma(z)\, y])\ \mathit{in}\ x.\mathrm{m} \\ b_2 & \triangleq & \mathit{let}\ y = \mathit{true}\ \mathit{in}\ (\mathit{let}\ x = [\mathrm{m} = \varsigma(z)\, y]\ \mathit{in}\ x.\mathrm{m}) \\ b_3 & \triangleq & \mathit{let}\ x = (\mathit{let}\ y = \mathit{true}\ \mathit{in}\ [\mathrm{f} = y,\ \mathrm{m} = \varsigma(z)\, z.\mathrm{f}])\ \mathit{in}\ x.\mathrm{m} \end{array}$$

All three programs are well-typed and yield the result *true*. Using our rules, we can prove $\emptyset \Vdash b_2 : \mathit{Bool} :: (r = \mathit{true})$ and $\emptyset \Vdash b_3 : \mathit{Bool} :: (r = \mathit{true})$ but not $\emptyset \Vdash b_1 : \mathit{Bool} :: (r = \mathit{true})$. A reasonable diagnosis is that the judgment $E \Vdash a : A :: T$ does not allow sufficient interaction between A and T (particularly in the rule for *let*). One remedy is transforming b_1 into b_2 (by let-floating [PPS96]) or into b_3 (by adding an auxiliary field). We have considered other remedies, but do not yet know which is the "right" one.

A deeper difficulty arises because the verification rules rely on a "global store" model. As Meyer and Sieber have explained [MS88], the use of this model is a source of incompleteness for procedural languages with local variables. Some of their remarks apply to our language as well. For example, the following program is reminiscent of their Example 2: $\mathit{let}\ x = [\mathrm{f} = \mathit{true}]\ \mathit{in}\ (y.\mathrm{m}\ ;\ x.\mathrm{f})$. This program will always return *true* because the method invocation y.m cannot affect the field f of the newly allocated object x. We can prove this, but only by adopting a strong specification for y, for example requiring that y.m not modify the field f of any object. Recently, there has been progress in the semantics of procedural languages with local variables (e.g., see [OT95, PS93]). Some of the insights gained in that area should be applicable to reasoning about objects.

6 Past and future work

As we mentioned in the introduction, there has been much research on specification and verification for object-oriented languages. The words "object" and "logic" are frequently used together in the literature, but with many different meanings (e.g., [SSC95]). Our work is most similar to that of Leavens [Lea89], who gave verification rules for a small language with objects; however, those rules are limited in that they apply only to programs without side-effects and aliasing. We do not know of any previous Hoare logic for a language like ours.

Much of the emphasis of the previous research has been on issues of refinement and inheritance. Lano and Haughton [LH92], Leavens [Lea89, Lea91], and Liskov and Wing [LW94] all studied notions of subtyping and of refinement of specifications (similar to our subspecification relation, though in some respects more sophisticated). Stata and Guttag [SG95] studied the notion of subclassing, and presented a pre-formal approach for reasoning about inheritance. Lano and Haughton [LH94] have collected other research on object-oriented specification.

In some existing formalisms (e.g., Leavens'), specifications can be written in terms of abstract variables. Specifications at different levels of abstraction can be related by simulation relations or abstraction functions. Undoubtedly the use of abstraction is important for specification and verification. We leave a full treatment of abstraction for future work; some results on abstraction appear in Leino's dissertation [Lei95], which also includes a guarded-command semantics for objects.

Several other extensions to our logic might be interesting. For example, it would be trivial to account for a construct that compares the addresses of two objects, or for a cloning construct. Recursive types and recursive specifications would be helpful in dealing with programs that manipulate unbounded object data structures, which our logic treats only in a limited way. The addition of concurrency primitives would be more difficult; it would call for a change of formalism, similar to the move from Hoare logic to Owicki-Gries logic [OG76].

7 Conclusions

In summary, the main outcome of our work is a logic that enables us (at least in principle) to specify and to verify object-oriented programs. To our knowledge, our notations and rules are novel. They permit proofs that, despite their simplicity, are outside the scope of previous methods. However, our work is only a first step; we hope that it stimulates further research.

Secondarily, we hope that our logic will serve as another datapoint on the relations between types and specifications. In the realm of functional programming, specifications can be seen as a neat generalization of ordinary types (through notions such as dependent types, or in the context of abstract interpretations). In our experience with imperative object-oriented languages, the step from types to specifications is not straightforward; still, type theory is sometimes helpful, for example in suggesting techniques for soundness proofs.

Acknowledgments Luca Cardelli and Greg Nelson helped in the initial stages of this work. Gary Leavens and Raymie Stata told us about related research. Luca Cardelli, Rowan Davies, and anonymous referees made useful comments on drafts of this paper.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, New York, 1996.

[Apt81] K.R. Apt. Ten years of Hoare's logic: A survey—Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Cla79] E.M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. *Journal of the ACM*, 26(1):129–147, January 1979.

[Flo67] R.W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.

[Har94] R. Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Jon92] C.B. Jones. An object-based design method for concurrent programs. Technical Report UMCS-92-12-1, University of Manchester, 1992.

[Lea89] G.T. Leavens. *Verifying Object-Oriented Programs that Use Subtypes.* PhD thesis, MIT Laboratory for Computer Science, February 1989. Available as Technical Report MIT/LCS/TR-439.

[Lea91] G.T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, pages 72–80, July 1991.

[Lei95] K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Ler92] X. Leroy. Polymorphic typing of an algorithmic language. Technical report, Institut National de Recherche en Informatique et en Automatique, October 1992. English version of the author's PhD thesis.

[LH92] K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, pages 78–97. Springer-Verlag LNCS 615, June 1992.

[LH94] K. Lano and H. Haughton. *Object-Oriented Specification Case Studies.* Prentice Hall, New York, 1994.

[LW94] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[MS88] A.R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 191–203, January 1988.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[OT95] P.W. O'Hearn and R.D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.

[PPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 1–12, May 1996.

[PS93] A.M. Pitts and I.D.B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

[SG95] R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. *ACM SIGPLAN Notices*, 30(10):200–214, October 1995. OOPSLA '95 conference proceedings.

[SSC95] A. Sernadas, C. Sernadas, and J.F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, November 1990.

[WF94] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

[YT87] A. Yonezawa and M. Tokoro, editors. *Object-oriented Concurrent Programming*. MIT Press, 1987.