September 15, 1998

SRC Research Report

161

A logic of object-oriented programs

Martín Abadi
and
K. Rustan M. Leino

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

A logic of object-oriented programs

Martín Abadi and K. Rustan M. Leino

September 15, 1998

A preliminary version of this work appeared in the proceedings of TAP-SOFT '97, volume 1214 of Lecture Notes in Computer Science, Springer, 1997. ©Springer 1997.

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.

Contents

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. However, the language lacks many class-based constructs common in practice. It also lacks "advanced" features, like concurrency; some of these features have been studied in the literature (e.g., see [dB90, 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:

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

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.

- $\mathit{if}\, x\, \mathit{then}\, a_0\, \mathit{else}\, a_1$ evaluates a_0 if x is *true* and evaluates a_1 if x is *false*.

- $\mathit{let}\ x = a\ \mathit{in}\ b$ evaluates a and then evaluates b with x bound to the result of a. We define $a\ ;\ b$ as a shorthand for $\mathit{let}\ x = a\ \mathit{in}\ b$ 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

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

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.

In order to formalize the operational semantics, we use some notations for partial functions. We write $A \rightharpoonup B$ for the set of partial functions from A to B. We write $\emptyset$ for the totally undefined partial function. When $f \in A \rightharpoonup B$, $a \in A$, and $b \in B$, we write $f.(a \mapsto b)$ for the function that coincides with f except possibly at a, and that maps a to b. When $a_i \in A\ {}^{i\in 1..n}$ are distinct and $b_i \in B\ {}^{i\in 1..n}$, we write $(a_i \mapsto b_i\ {}^{i\in 1..n})$ for the function in $A \rightharpoonup B$ that maps a_i to b_i for $i \in 1..n$ and is otherwise undefined.

The formal operational semantics is in terms of stacks and stores. A stack maps variables to booleans or references. A store maps object fields to booleans or references and maps object methods to closures. We write $\sigma, S \vdash b \rightsquigarrow v, \sigma'$ to mean that, given the initial store σ and the stack S, executing the program b leads to the result v and to the final store σ'.

We define the notions of store, stack, and result as follows:

- We assume we are given a set of *object names* $\mathcal{H}$. The set of *results* $\mathcal{R}$ is $\mathcal{H} \cup \{\mathit{false}, \mathit{true}\}$.
- A *stack* is a function in $\mathcal{V} \rightharpoonup \mathcal{R}$.
- A *method closure* is a triple of a variable $x \in \mathcal{V}$ (standing for self), a program b, and a stack S; we write it $\langle\varsigma(x)b, S\rangle$. The set of method closures is $\mathcal{C}$.
- A *store* is a function σ in $\mathcal{H} \rightharpoonup ((\mathcal{F} \cup \mathcal{M}) \rightharpoonup (\mathcal{R} \cup \mathcal{C}))$. There is a condition on σ: if $h \in \mathcal{H}$, $\mathrm{f} \in \mathcal{F}$, and $\sigma(h)(\mathrm{f})$ is defined, then $\sigma(h)(\mathrm{f}) \in \mathcal{R}$; if $h \in \mathcal{H}$, $\mathrm{m} \in \mathcal{M}$, and $\sigma(h)(\mathrm{m})$ is defined, then $\sigma(h)(\mathrm{m}) \in \mathcal{C}$. In other words, field names are mapped to results and method names to closures.

The operational semantics is represented with a set of rules, given below. According to these rules, a variable x reduces to its value in the stack, without change in the store. The constants *false* and *true* reduce to themselves, without change in the store. The execution of a conditional expression consists in evaluating the guard and, depending on the outcome of this evaluation, evaluating one of the branches. The *let* construct evaluates an expression, binds a local variable to the result of that evaluation, and then evaluates another expression. The execution of an object construction requires evaluating the fields, constructing method closures, picking a new location, and mapping that location to an appropriate suite of fields and methods. The execution of a field selection on an object requires evaluating the object and then extracting the value of the appropriate field from the store. The execution of a method invocation is similar, but there the value returned is the result of evaluating the appropriate method body with an extended stack that maps the self variable to the value of the object. Finally, the execution of a field update modifies the store and returns the value of the object being affected.

Operational semantics

Variables

$$\frac{S(x) = v}{\sigma, S \vdash x \leadsto v, \sigma}$$

Constants

$$\frac{}{\sigma, S \vdash \mathit{false} \leadsto \mathit{false}, \sigma} \qquad \frac{}{\sigma, S \vdash \mathit{true} \leadsto \mathit{true}, \sigma}$$

Conditional

$$\frac{S(x) = \mathit{false} \qquad \sigma, S \vdash a_1 \leadsto v, \sigma'}{\sigma, S \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 \leadsto v, \sigma'}$$

$$\frac{S(x) = \mathit{true} \qquad \sigma, S \vdash a_0 \leadsto v, \sigma'}{\sigma, S \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 \leadsto v, \sigma'}$$

Let

$$\frac{\sigma, S \vdash a \leadsto v, \sigma' \qquad \sigma', S.(x \mapsto v) \vdash b \leadsto v', \sigma''}{\sigma, S \vdash \mathit{let}\ x = a\ \mathit{in}\ b \leadsto v', \sigma''}$$

Object construction

$$\frac{\begin{array}{l} S(x_i) = v_i\ ^{i \in 1..n} \qquad h \notin \mathit{dom}(\sigma) \qquad h \in \mathcal{H} \\ \sigma' = \sigma.(h \mapsto (\mathrm{f}_i \mapsto v_i\ ^{i \in 1..n},\ \mathrm{m}_j \mapsto \langle \varsigma(y_j)b_j, S \rangle\ ^{j \in 1..m})) \end{array}}{\sigma, S \vdash [\mathrm{f}_i = x_i\ ^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j)b_j\ ^{j \in 1..m}] \leadsto h, \sigma'}$$

Field selection

$$\frac{S(x) = h \qquad h \in \mathcal{H} \qquad \sigma(h)(\mathrm{f}) = v}{\sigma, S \vdash x.\mathrm{f} \leadsto v, \sigma}$$

Method invocation

$$\frac{\begin{array}{l} S(x) = h \qquad h \in \mathcal{H} \qquad \sigma(h)(\mathrm{m}) = \langle \varsigma(y)b, S' \rangle \\ \sigma, S'.(y \mapsto h) \vdash b \leadsto v, \sigma' \end{array}}{\sigma, S \vdash x.\mathrm{m} \leadsto v, \sigma'}$$

Field update

$$\frac{\begin{array}{l} S(x) = h \qquad h \in \mathcal{H} \qquad \sigma(h)(\mathrm{f}) \text{ is defined} \\ S(y) = v \qquad \sigma' = \sigma.(h \mapsto \sigma(h).(\mathrm{f} \mapsto v)) \end{array}}{\sigma, S \vdash x.\mathrm{f} := y \leadsto h, \sigma'}$$

The judgment $\sigma, S \vdash b \leadsto v, \sigma'$ represents only computations that terminate with a result, not computations that do not terminate or that terminate with an error. For example, intuitively, the execution of $let\ x = [\mathrm{m} = \varsigma(y)\ true]\ in\ x.\mathrm{m}$ terminates with the output $true$. Formally, we can derive

$$\sigma, S \vdash let\ x = [\mathrm{m} = \varsigma(y)\ true]\ in\ x.\mathrm{m} \leadsto true, \sigma'$$

for all σ and S and for some σ'. On the other hand, intuitively, the execution of $let\ x = true\ in\ x.\mathrm{m}$ yields an error, while the execution of $let\ x = [\mathrm{m} = \varsigma(x)\ x.\mathrm{m}]\ in\ x.\mathrm{m}$ does not terminate. Formally,

$$\sigma, S \vdash let\ x = true\ in\ x.\mathrm{m} \leadsto v, \sigma'$$

and

$$\sigma, S \vdash let\ x = [\mathrm{m} = \varsigma(x)\ x.\mathrm{m}]\ in\ x.\mathrm{m} \leadsto v, \sigma'$$

cannot be derived for any σ, S, v, and σ'. The search for a derivation of the former judgment "gets stuck", while the search for a derivation of the latter judgment diverges.

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:

$$[\mathrm{f}_i{:}\, A_i\ ^{i \in 1..n},\ \mathrm{m}_j{:}\, B_j\ ^{j \in 1..m}]$$

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 $[\mathrm{f}_i{:}\, A_i\ ^{i \in 1..n+p},\ \mathrm{m}_j{:}\, B_j\ ^{j \in 1..m+q}]$ is a subtype of $[\mathrm{f}_i{:}\, A_i\ ^{i \in 1..n},\ \mathrm{m}_j{:}\, B'_j\ ^{j \in 1..m}]$ 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

$$\frac{}{\vdash \mathit{Bool}} \qquad \frac{\vdash A_i \;{}^{i \in 1..n} \qquad \vdash B_j \;{}^{j \in 1..m}}{\vdash [\mathrm{f}_i \colon A_i \;{}^{i \in 1..n},\ \mathrm{m}_j \colon B_j \;{}^{j \in 1..m}]}$$

Subtypes

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

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

A typing environment is a (possibly empty) list of pairs $x\colon A$, 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 $x\colon A$ 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

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

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 $\stackrel{\mathit{syn}}{=}$ 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

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

Constants

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

Conditional

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

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

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

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

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

This type system is much 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 (as in the work of Liskov and Wing [LW94], for example). However, as the next section explains, specifications are a generalization of 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 postconditions. 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, $\grave{alloc}(x)$ and $\acute{alloc}(x)$ 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 $r = \grave{\sigma}(x, \mathrm{f}) \ \wedge \ (\forall y, z \,.\, \grave{\sigma}(y, z) = \acute{\sigma}(y, z) \ \wedge \ (\grave{alloc}(y) \ \equiv \ \acute{alloc}(y))\ )$.

We work in standard first-order logic, so the functions $\grave{\sigma}$ and $\acute{\sigma}$ are total. Hence, $\grave{\sigma}(x, \mathrm{f})$ is defined even if $\grave{alloc}(x)$ does not hold, and $\acute{\sigma}(x, \mathrm{f})$ is defined even if $\acute{alloc}(x)$ does not hold. In those cases, the values of $\grave{\sigma}(x, \mathrm{f})$ and $\acute{\sigma}(x, \mathrm{f})$ are not important. Similarly, the values of expressions such as $\grave{\sigma}(\mathrm{f}, x)$ and $\grave{alloc}(\mathrm{f})$, which are intuitively meaningless, 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 $\grave{alloc}$ and $\acute{alloc}$;
- constants for field names (such as f);
- other variables (such as x);
- the usual logical connectives $\neg$, $\wedge$, and $\forall$, and the equality predicate $=$ (from which $\vee$, $\Rightarrow$, $\equiv$, $\exists$, and $\neq$ can be defined as abbreviations).

The grammar for transition relations is thus:

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

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. Informally, 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. 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

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

Subspecifications

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

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

In this last rule, $\Vdash_{fol}$ represents provability in first-order logic with the standard axioms for $=$ and the axioms $\mathit{false} \neq \mathit{true}$ and $\mathrm{f} \neq \mathrm{g}$ for every pair of different field names f and g.

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 rules of the operational semantics (section 2.1) and with the typing rules (section 2.2).

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; it generalizes both the subsumption rule for typing and the standard rule of consequence from Hoare logic. 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 $\check{alloc}$. (Note that the grammar for transition relations does not allow the auxiliary symbols $\check{\sigma}$ and $\check{alloc}$; however, $\Vdash_{fol}$ applies to any first-order formula, even one that is not a transition relation.) 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:

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

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

$$\frac{\Vdash A <: A' \qquad \Vdash_{fol} T \Rightarrow T' \qquad E \Vdash a : A :: T \qquad E \Vdash A' \qquad E \Vdash T' \text{ is a transition relation}}{E \Vdash a : A' :: T'}$$

Variables

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

Constants

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

Conditional

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

Let

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

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

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

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

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

4 Examples

We discuss a few instructive examples, some of them with derivations. From now on, we use some abbreviations, allowing general expressions to appear where the grammar requires a variable. In case a, $a_i\ ^{i \in 1..n}$, and b are not variables, we define:

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

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. For example, for field selection, we may use the rule:

$$\frac{E \Vdash a : [\mathrm{f} \colon A] :: T}{E \Vdash a.\mathrm{f} : A :: (\ \exists x\,.\ T[x/r] \wedge \mathit{Res}(\acute{\sigma}(x, \mathrm{f}))\ )}$$

4.1 Field update and selection

Our first example concerns the program:

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

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 $r = \mathit{true}$ holds upon termination of this program. Formally, we can derive the judgment:

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

4.2 Aliasing

The following three programs exhibit the rôle of aliasing:

$$\mathit{let}\ x = [\mathrm{f} = \mathit{false}]\ \mathit{in}\ \mathit{let}\ y = [\mathrm{g} = \mathit{false}]\ \mathit{in}\ (y.\mathrm{g} := \mathit{true}\ ;\ x.\mathrm{f})$$

$$\mathit{let}\ x = [\mathrm{f} = \mathit{false}]\ \mathit{in}\ \mathit{let}\ y = [\mathrm{f} = \mathit{false}]\ \mathit{in}\ (y.\mathrm{f} := \mathit{true}\ ;\ x.\mathrm{f})$$

$$\mathit{let}\ x = [\mathrm{f} = \mathit{true}]\ \mathit{in}\ \mathit{let}\ y = x\ \mathit{in}\ (y.\mathrm{f} := \mathit{false}\ ;\ x.\mathrm{f})$$

For each of these programs we can verify that $r = \mathit{false}$. 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.

4.3 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:

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

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:

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

(The proof relies on the addition of standard axioms about integers to the underlying first-order logic.) In verifying the body of the method m, we can use the specification of m, recursively. The derivation included in the next example demonstrates how a method specification can be used recursively in a formal proof.

4.4 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 specification A and transition relation T without free variables.

We show the proof as a sequence of judgments, from the desired conclusion back to "true", indicating between braces the rules applied and other justifications. We write *False* and *True* as abbreviations for the formulas (*false* $=$ *true*) and (*false* $=$ *false*), respectively.

$\quad\emptyset \Vdash [\mathrm{m} = \varsigma(x)\, x.\mathrm{m}].\mathrm{m} : A :: T$
$\Leftarrow \qquad \{$ subsumption, using the assumptions and $\Vdash_{fol}$ *False* $\Rightarrow T$ $\}$
$\quad\emptyset \Vdash [\mathrm{m} = \varsigma(x)\, x.\mathrm{m}].\mathrm{m} : A :: \mathit{False}$
$\equiv \qquad \{$ shorthand $\}$
$\quad\emptyset \Vdash \mathit{let}\ x = [\mathrm{m} = \varsigma(x)\, x.\mathrm{m}]\ \mathit{in}\ x.\mathrm{m} : A :: \mathit{False}$
$\Leftarrow \qquad \{$ let $\}$
$\quad\emptyset \Vdash [\mathrm{m} = \varsigma(x)\, x.\mathrm{m}] : [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] :: \mathit{True}$
$\quad\emptyset, x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] \Vdash x.\mathrm{m} : A :: \mathit{False}$
$\quad\emptyset \Vdash A$
$\quad E \Vdash \mathit{False}$ is a transition relation
$\quad\Vdash_{fol} \mathit{True} \wedge \mathit{False} \Rightarrow \mathit{False}$
$\Leftarrow \qquad \{$ assumptions and first-order logic $\}$
$\quad\emptyset \Vdash [\mathrm{m} = \varsigma(x)\, x.\mathrm{m}] : [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] :: \mathit{True}$
$\quad\emptyset, x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] \Vdash x.\mathrm{m} : A :: \mathit{False}$
$\Leftarrow \qquad \{$ subsumption; object construction $\}$
$\quad\emptyset \Vdash \diamond$
$\quad\emptyset, x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] \Vdash x.\mathrm{m} : A :: \mathit{False}$
$\quad\emptyset, x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] \Vdash x.\mathrm{m} : A :: \mathit{False}$
$\Leftarrow \qquad \{$ well-formed specification environments; simplification $\}$
$\quad\emptyset, x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] \Vdash x.\mathrm{m} : A :: \mathit{False}$
$\Leftarrow \qquad \{$ method invocation $\}$
$\quad\emptyset, x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] \Vdash x : [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] :: \mathit{Res}(x)$
$\Leftarrow \qquad \{$ variables $\}$
$\quad\emptyset, x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] \Vdash \diamond$
$\Leftarrow \qquad \{$ well-formed specification environments $\}$
$\quad\emptyset \Vdash \diamond$
$\quad\Vdash A$
$\quad$ *False* is a transition relation
$\Leftarrow \qquad \{$ well-formed specification environments; assumptions $\}$
$\quad$ true

Note that in the step that uses the rule for object construction we derive $\emptyset \Vdash [\mathrm{m} = \varsigma(x)\, x.\mathrm{m}] : [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] :: \mathit{True}$ from $\emptyset, x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}] \Vdash x.\mathrm{m} : A :: \mathit{False}$. The assumption $x{:}\, [\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}]$ then enables us to check that the method body $x.\mathrm{m}$ implements the specification for m given in $[\mathrm{m}{:}\, \varsigma(x)A :: \mathit{False}]$. Thus, recursively, we use the specification of an object in verifying its implementation.

4.5 Putting it all together

Let a_0 be the following object:

$$[\ \mathrm{f} = \mathit{false},\ \ \mathrm{m} = \varsigma(s)\ \mathit{if}\ s.\mathrm{f}\ \mathit{then\, false\ else\ true}\ ]$$

Below we give a detailed proof that this object satisfies the following specification:

$$\begin{array}{ll} [\ \mathrm{f}{:}\, \mathit{Bool}, & \\ \ \ \mathrm{m}{:}\, \varsigma(s)\ \mathit{Bool} :: & (\grave{\sigma}(s,\mathrm{f}) = \mathit{false} \Rightarrow r = \mathit{true}) \wedge \\ & (\grave{\sigma}(s,\mathrm{f}) = \mathit{true} \Rightarrow r = \mathit{false})\ ] \end{array}$$

The specification says that the object's m method returns the negation of the object's f field. Below we also give a detailed proof of the following judgment, illustrating the use of a_0:

$$\emptyset \Vdash \mathit{let}\ x = a_0\ \mathit{in}\ (x.\mathrm{f} := \mathit{true}\ ;\ x.\mathrm{m}) : \mathit{Bool} :: r = \mathit{false}$$

The proofs include applications of all our verification rules.

For convenience in the proofs, we define a transition relation T, as follows:

$$T \ \triangleq\ (\grave{\sigma}(s,\mathrm{f}) = \mathit{false} \Rightarrow r = \mathit{true}) \wedge (\grave{\sigma}(s,\mathrm{f}) = \mathit{true} \Rightarrow r = \mathit{false})$$

We also define A_0 to be the specification of a_0:

$$A_0 \ \triangleq\ [\ \mathrm{f}{:}\, \mathit{Bool},\ \mathrm{m}{:}\, \varsigma(s)\mathit{Bool} :: T\ ]$$

We begin with a proof of the specification of a_0, more precisely, with a proof of the judgment $E \Vdash a_0 : A_0 :: r = r$. We omit some easy steps, and use our standard abbreviations. For any well-formed environment E not containing s, we calculate:

$$\begin{array}{ll} & E \Vdash a_0 : A_0 :: r = r \\ \Leftarrow & \qquad \{\ \ \text{subsumption}\ \ \} \end{array}$$

$$
\begin{array}{ll}
& E \Vdash a_0 : A_0 :: \\
& \qquad \neg\grave{alloc}(r) \wedge \acute{alloc}(r) \wedge (\,\forall z \,.\, z \neq r \Rightarrow (\grave{alloc}(z) \equiv \acute{alloc}(z))\,) \wedge \\
& \qquad \acute{\sigma}(r, \mathrm{f}) = \mathit{false} \wedge (\,\forall z, w \,.\, z \neq r \Rightarrow \grave{\sigma}(z, w) = \acute{\sigma}(z, w)\,) \\
\Leftarrow & \qquad \{ \ \text{object construction} \ \} \\
& E \Vdash \diamond \\
& E \Vdash \mathit{false} : \mathit{Bool} :: \mathit{Res}(\mathit{false}) \\
& E, s{:}\, A_0 \Vdash \mathit{if}\ s.\mathrm{f}\ \mathit{then}\ \mathit{false}\ \mathit{else}\ \mathit{true} : \mathit{Bool} :: T \\
\Leftarrow & \qquad \{ \ E \text{ is well-formed; constants} \ \} \\
& E, s{:}\, A_0 \Vdash \mathit{if}\ s.\mathrm{f}\ \mathit{then}\ \mathit{false}\ \mathit{else}\ \mathit{true} : \mathit{Bool} :: T \\
\Leftarrow & \qquad \{ \ \text{conditional} \ \} \\
& E, s{:}\, A_0 \Vdash s.\mathrm{f} : \mathit{Bool} :: \mathit{Res}(\grave{\sigma}(s, \mathrm{f})) \\
& E, s{:}\, A_0 \Vdash \mathit{false} : \mathit{Bool} :: \\
& \qquad (\mathit{true} = \mathit{false} \Rightarrow r = \mathit{true}) \wedge (\mathit{true} = \mathit{true} \Rightarrow r = \mathit{false}) \\
& E, s{:}\, A_0 \Vdash \mathit{true} : \mathit{Bool} :: \\
& \qquad (\mathit{false} = \mathit{false} \Rightarrow r = \mathit{true}) \wedge (\mathit{false} = \mathit{true} \Rightarrow r = \mathit{false}) \\
\Leftarrow & \qquad \{ \ \text{field selection; subsumption, using } \neg(\mathit{false} = \mathit{true}) \ \} \\
& E, s{:}\, A_0 \Vdash s : A_0 :: \mathit{Res}(s) \\
& E, s{:}\, A_0 \Vdash \mathit{false} : \mathit{Bool} :: \mathit{Res}(\mathit{false}) \\
& E, s{:}\, A_0 \Vdash \mathit{true} : \mathit{Bool} :: \mathit{Res}(\mathit{true}) \\
\Leftarrow & \qquad \{ \ \text{variables; constants} \ \} \\
& E, s{:}\, A_0 \Vdash \diamond \\
\Leftarrow & \qquad \{ \ \text{well-formed specification environments} \ \} \\
& E \Vdash \diamond \\
& E \Vdash A_0 \\
& s \text{ not in } E \\
\Leftarrow & \qquad \{ \ E \text{ is well-formed and does not contain } s\text{; misc. rules} \ \} \\
& \text{true}
\end{array}
$$

Next, we prove that the program $\mathit{let}\ x = a_0\ \mathit{in}\,(x.\mathrm{f} := \mathit{true}\ ;\ x.\mathrm{m})$, which contains a_0, yields the result *false*:

$$
\begin{array}{ll}
& \emptyset \Vdash \mathit{let}\ x = a_0\ \mathit{in}\,(x.\mathrm{f} := \mathit{true}\ ;\ x.\mathrm{m}) : \mathit{Bool} :: r = \mathit{false} \\
\Leftarrow & \qquad \{ \ \text{let} \ \} \\
& \emptyset \Vdash a_0 : A_0 :: r = r \\
& \emptyset, x{:}\, A_0 \Vdash x.\mathrm{f} := \mathit{true}\ ;\ x.\mathrm{m} : \mathit{Bool} :: r = \mathit{false} \\
& \emptyset \Vdash \mathit{Bool} \\
& \emptyset \Vdash (r = \mathit{false}) \text{ is a transition relation} \\
& \Vdash_{\mathit{fol}} x = x \wedge r = \mathit{false} \Rightarrow r = \mathit{false} \\
\Leftarrow & \qquad \{ \ \text{previous proof about } a_0\text{; misc. rules} \ \} \\
& \emptyset, x{:}\, A_0 \Vdash x.\mathrm{f} := \mathit{true}\ ;\ x.\mathrm{m} : \mathit{Bool} :: r = \mathit{false} \\
\Leftarrow & \qquad \{ \ \text{definition of } ; \ \}
\end{array}
$$

$$
\begin{array}{ll}
& \emptyset, x{:}\,A_0 \Vdash \mathit{let}\ z = x.\mathrm{f} := \mathit{true}\ \mathit{in}\ x.\mathrm{m} : \mathit{Bool} :: r = \mathit{false} \\
\Leftarrow & \qquad \{\ \ \text{let}\ \ \} \\
& \emptyset, x{:}\,A_0 \Vdash x.\mathrm{f} := \mathit{true} : A_0 :: \acute{\sigma}(x,\mathrm{f}) = \mathit{true} \\
& \emptyset, x{:}\,A_0, z{:}\,A_0 \Vdash x.\mathrm{m} : \mathit{Bool} :: \grave{\sigma}(x,\mathrm{f}) = \mathit{true} \Rightarrow r = \mathit{false} \\
& \emptyset, x{:}\,A_0 \Vdash \mathit{Bool} \\
& \emptyset, x{:}\,A_0 \Vdash (r = \mathit{false}) \text{ is a transition relation} \\
& \Vdash_{\mathit{fol}} \check{\sigma}(x,\mathrm{f}) = \mathit{true} \wedge (\check{\sigma}(x,\mathrm{f}) = \mathit{true} \Rightarrow r = \mathit{false}) \Rightarrow r = \mathit{false} \\
\Leftarrow & \qquad \{\ \ \text{misc. rules}\ \ \} \\
& \emptyset, x{:}\,A_0 \Vdash x.\mathrm{f} := \mathit{true} : A_0 :: \acute{\sigma}(x,\mathrm{f}) = \mathit{true} \\
& \emptyset, x{:}\,A_0, z{:}\,A_0 \Vdash x.\mathrm{m} : \mathit{Bool} :: \grave{\sigma}(x,\mathrm{f}) = \mathit{true} \Rightarrow r = \mathit{false} \\
\Leftarrow & \qquad \{\ \ \text{subsumption; field update; method invocation}\ \ \} \\
& \emptyset, x{:}\,A_0 \Vdash x : A_0 :: \mathit{Res}(x) \\
& \emptyset, x{:}\,A_0, z{:}\,A_0 \Vdash x : A_0 :: \mathit{Res}(x) \\
\Leftarrow & \qquad \{\ \ \text{variables; misc. rules}\ \ \} \\
& \text{true}
\end{array}
$$

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 formal sanity check for the verification rules. It also highlights a limitation of the verification rules: 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 $r = \mathit{true}$, 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, every well-typed program 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. As we prove next, 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*. We call this property soundness.

A special case of our soundness theorem is:

Theorem 1 *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, $\emptyset, \emptyset \vdash b \rightsquigarrow v, \sigma'$ is provable with the rules of section 2.1 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.*

The statement of this theorem clearly expresses the consistency of the verification rules with the operational semantics. It is however unsatisfactory in at least two respects: (i) it does not apply to programs with free variables, to programs that return objects, or to programs that start running with a nonempty store; (ii) it cannot be proved directly anyway.

Going a little beyond the first special case, we can show that if b yields result v when run with an empty stack and an empty initial store, and if $\emptyset \Vdash b : A :: T$ is provable, then v "satisfies" A, and T holds when interpreted as a predicate on the initial and the final stores, with v as the value of r.

The full statement of our soundness theorem is as follows.

Theorem 2 *Assume that $\sigma, S \vdash a \rightsquigarrow v, \sigma'$ is provable and that $\Sigma \models \sigma$. If $E \Vdash a : A :: T$ is provable and $\Sigma \models S : E$, then $(S, \sigma, \sigma', v) \models T$ and there exists Σ' such that $\Sigma' \succeq \Sigma$, $\Sigma' \models \sigma'$, and $\Sigma' \models v : (A, S)$.*

The notations used in this statement are defined precisely in the appendix, where we also prove the theorem. Here we explain these notations only informally. The hypothesis $\Sigma \models \sigma$ means that the store σ meets the store specification Σ. The hypothesis $\Sigma \models S : E$ means that the variables in the stack S meet the specifications given in E in the context of Σ. The store specification Σ is needed because the initial store σ may be nonempty; similarly, the use of E as a specification for the stack S is needed because the program b may have free variables. The conclusion $(S, \sigma, \sigma', v) \models T$ means that the transition relation holds when we interpret its symbols using S, σ, σ', and v (for example, taking v as the value of r). The conclusions $\Sigma' \succeq \Sigma$ and $\Sigma' \models \sigma'$ imply that Σ' agrees with Σ but possibly specifies additional objects in the store (those allocated in going from σ to σ'); these two conclusions appear in order to permit a direct inductive proof. Finally, the conclusion $\Sigma' \models v : (A, S)$ means that the output v meets the specification A in the context of Σ' and S.

Theorem 1 is a corollary of Theorem 2. As another corollary, we obtain a soundness theorem for the type system of section 2.2. Therefore, as might be expected, our soundness proof is no less intricate than proofs of type soundness for imperative languages. In fact, Theorem 2 generalizes concepts recently developed for sophisticated proofs of type soundness [AC96, Har94, Ler92, Tof90, WF94]. New techniques 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:

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

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]). We do not know of any previous Hoare logic for a language like ours.

Our work is most similar to that of Leavens [Lea89], who developed a Hoare logic for a small language with objects. The language is statically typed and includes a subtyping relation, but does not permit side-effects or aliasing. In another related study, de Boer [dB90] gave verification rules for the parallel language POOL. These rules apply to programs with side-effects and aliasing, but without subtyping or recursive methods, and with only one global implementation for each method (rather than one implementation per object). Both Leavens and de Boer obtained soundness results. Recently, in his dissertation [PH97], Poetzsch-Heffter considered how to integrate Larch-style interface specifications with Hoare-style verification techniques for object-oriented programs. Further, Poetzsch-Heffter and Müller [PHM98] provided a proof system for a class-based language, representing properties of the type system of the language in axioms.

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's), 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, and probably an essential ingredient of any practical programming tool. We leave a full treatment of abstraction for future work; some results on abstraction appear in Leino's dissertation [Lei95], which also gives a guarded-command semantics for objects and uses this semantics for reasoning about programs.

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; a recent continuation of this work considers recursion [Lei98]. Rules for subclasses and inheritance would be important for treating standard class-based languages like Modula-3 and Java; perhaps one could develop a formal version of Stata and Guttag's approach. It remains to be seen whether class-based constructs should be included explicitly or whether their simulation in terms of simpler object-based constructs provides a sufficiently convenient basis for reasoning. 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. Krzysztof Apt, Frank de Boer, Luca Cardelli, Rowan Davies, and David Naumann made useful comments on drafts of this paper.

Appendix: Soundness (definitions and theorems)

The soundness theorem requires several auxiliary notions. Before giving their definitions, we motivate each of them informally.

As part of the soundness theorem, we show that if the verification rules say that a satisfies the specification A and if, operationally, a yields the result v, then v satisfies A. Making precise the assertion that v satisfies A is delicate for several reasons.

One problem is that program variables may occur free in A. We surmount this difficulty by considering specification closures. A specification closure is a specification paired with a stack that gives values for the free program variables of the specification. (See Definition 1.) Instead of saying that v satisfies A, we can say that v satisfies the specification closure (A, S), where S is the stack used for the execution of a.

Definition 1 (Specification closures)

- *A specification closure is a pair* (A, S) *where* A *is a specification and* S *is a stack such that the free program variables of* A *are all in the domain of* S.

- *Given a specification* A *and a stack* S*, we write* AS *for the result of replacing the free program variables of* A *by the corresponding results from* S*. Similarly, we write* TS *when* T *is a formula.*

- *We handle* AS *and* TS *as formal expressions by treating all object names as free variables. In particular, we may write the subtyping assertion* $A'S' <: AS$*; this assertion is defined by the standard rules for subtyping. (We omit a* $\Vdash$ *in* $A'S' <: AS$ *in order to stress that this is not a judgment of our verification system.)*

A second problem is that v may be an address in the store (that is, an object name) and the store may contain cycles; cycles impede inductive definitions. We surmount this difficulty by introducing store specifications, which associate specification closures with object names.

Definition 2 (Store specifications)

- *A store specification is a partial function that maps object names (from the set* $\mathcal{H}$*) to specification closures.*

- *Given store specifications* Σ *and* Σ'*, we write* $\Sigma' \succeq \Sigma$ *if* Σ' *extends* Σ*.*

- *Given a store specification Σ, a specification closure (A, S), and a result v, $\Sigma \models_0 v : (A, S)$ holds if either A is Bool and v is one of false and true, or if (A, S) is $\Sigma(v)$ and $v \in \mathcal{H}$.*

- *$\Sigma \models v : (B, S)$ holds if there exist B' and S' such that $B'S' <: BS$ and $\Sigma \models_0 v : (B', S')$.*

The verification of a takes place with a particular specification environment, and the execution of a takes place with a particular stack. The soundness theorem assumes that the stack matches the environment, in the sense that if v_i is the value for x_i in the stack, and A_i is its specification in the environment, then v_i satisfies a suitable specification closure (A_i, S_i).

Definition 3 (Stacks vs. environments) *The relation $\Sigma \models S : E$ is defined inductively by:*

- *If Σ is a store specification, then $\Sigma \models \emptyset : \emptyset$.*

- *If $\Sigma \models S : E$, $\Sigma \models v : (A, S)$, and x is not in E, then $\Sigma \models S.(x \mapsto v) : (E, x{:}\, A)$.*

For a store specification to be useful, it needs to be consistent with the particular store under consideration. We define this consistency relation without breaking cycles in the store, as follows.

Definition 4 (Stores vs. store specifications) *Given a store σ and a store specification Σ, $\Sigma \models \sigma$ holds if Σ and σ have the same domain, and for every v in their domain, $\Sigma(v)$ has the form (A, S) where A is $[\mathrm{f}_i{:}\, A_i \ ^{i \in 1..n},\ \mathrm{m}_j{:}\, \varsigma(y_j)B_j :: T_j \ ^{j \in 1..m}]$, and*

- *for $i \in 1..n$, $\sigma(v)(\mathrm{f}_i)$ is defined and $\Sigma \models \sigma(v)(\mathrm{f}_i) : (A_i, S)$;*

- *for $j \in 1..m$, $\sigma(v)(\mathrm{m}_j)$ is of the form $\langle \varsigma(y_j)b_j, S\rangle$, and $E, y_j{:}\, A \Vdash b_j : B_j :: T_j$ for some E such that $\Sigma \models S : E$.*

As part of the soundness theorem, we show also that if the verification rules say that a satisfies the transition relation T and if the execution of a with the initial store σ yields the store σ', then T is true when interpreted as a predicate on the stores σ and σ', with v as the value of r. Making precise the assertion that T is true is not too difficult, since T is simply a formula in first-order logic.

In order to interpret a first-order-logic formula, all one does is give a domain (a nonempty set), associate relations on this domain with predicate symbols, associate operations on this domain with function symbols, and map variables to elements of the domain [Bar77]. Collectively, the domain, the relations, and the operations are called a structure; the mapping of variables to elements is called an assignment. In the case of T, we define the structure from the stores σ and σ'; the assignment maps r to v, and maps any other free variables of T to their values in the stack.

Definition 5 (Satisfaction for formulas) *Given two stores σ and σ', we define a structure, as follows:*

- *The domain of the structure is $\{false, true\} \cup \mathcal{H} \cup \mathcal{F} \cup \{\perp\}$ (so it includes booleans, object names, field names, and a special, distinct undefined value).*
- *false, true, and all field names are interpreted as themselves.*
- *$\grave{\sigma}$ is interpreted as the binary function that maps any d_1 and d_2 to $\sigma(d_1)(d_2)$ if this is defined and to $\perp$ otherwise.*
- *$\acute{\sigma}$ is interpreted as the binary function that maps any d_1 and d_2 to $\sigma'(d_1)(d_2)$ if this is defined and to $\perp$ otherwise.*
- *$\grave{alloc}$ is interpreted as the unary predicate that maps any d to true if and only if d is in the domain of σ.*
- *$\acute{alloc}$ is interpreted as the unary predicate that maps any d to true if and only if d is in the domain of σ'.*

Given a stack S and a result v, we define an assignment of elements of the domain to variables, as follows:

- *r is interpreted as v.*
- *Any other variable x is interpreted as $S(x)$ if this is defined and as $\perp$ otherwise.*

Given a transition relation T, we write $(S, \sigma, \sigma', v) \models T$ if the assignment associated with S and v satisfies T in the structure associated with σ and σ'.

Two simple lemmas, given next without proof, state some of the properties of the notions defined above.

Lemma 1 (Substitution) *If A and B are specifications such that $\Vdash A <: B$, and S is a stack, then $AS <: BS$.*

Lemma 2 (Extension) *If* $\Sigma \models S : E$ *and* $\Sigma' \succeq \Sigma$*, then* $\Sigma' \models S : E$*.*

Theorem 1, given in the main body of this paper, is a special case of Theorem 2, which we restate and prove here.

Theorem 2 *Assume that* $\sigma, S \vdash a \rightsquigarrow v, \sigma'$ *is provable and that* $\Sigma \models \sigma$*. If* $E \Vdash a : A :: T$ *is provable and* $\Sigma \models S : E$*, then* $(S, \sigma, \sigma', v) \models T$ *and there exists* Σ' *such that* $\Sigma' \succeq \Sigma$*,* $\Sigma' \models \sigma'$*, and* $\Sigma' \models v : (A, S)$*.*

Proof The proof is a direct induction on the derivation of $\sigma, S \vdash a \rightsquigarrow v, \sigma'$. There is a case for each of the rules of the operational semantics. (Unlike in some proofs for Hoare logic [Apt81], the possibility of recursion does not lead to the consideration of approximations to recursive procedures.)

Variables

$$\frac{S(x) = v}{\sigma, S \vdash x \rightsquigarrow v, \sigma}$$

Suppose $\Sigma \models \sigma$, $\Sigma \models S : E$, and $E \Vdash x : A :: T$. Since $E \Vdash x : A :: T$, the environment E must contain $x{:}\,A'$ for some A' such that $\Vdash A' <: A$, and $\Vdash_{fol} Res(x) \Rightarrow T$. We obtain:

- $(S, \sigma, \sigma, v) \models T$, since $\Vdash_{fol} Res(x) \Rightarrow T$ and $(S, \sigma, \sigma, v) \models Res(x)$ (because $S(x) = v$).
- $\Sigma \succeq \Sigma$, trivially.
- $\Sigma \models \sigma$, by hypothesis.
- $\Sigma \models v : (A, S)$: Since $\Sigma \models S : E$, we must have $\Sigma \models v : (A', S')$ for some prefix S' of S, and hence $\Sigma \models_0 v : (A'', S'')$ for some (A'', S'') such that $A''S'' <: A'S'$. Since $\Vdash A' <: A$, the substitution lemma yields $A'S <: AS$. By transitivity, we obtain $A''S'' <: AS$, since $A'S' = A'S$. Therefore, $\Sigma \models v : (A, S)$.

Constants

$$\frac{}{\sigma, S \vdash \mathit{false} \rightsquigarrow \mathit{false}, \sigma} \qquad \frac{}{\sigma, S \vdash \mathit{true} \rightsquigarrow \mathit{true}, \sigma}$$

We argue the case for *false*. Suppose $\Sigma \models \sigma$, $\Sigma \models S : E$, and $E \Vdash \mathit{false} : A :: T$. Since $E \Vdash \mathit{false} : A :: T$, we must have $A = \mathit{Bool}$ and $\Vdash_{fol} Res(\mathit{false}) \Rightarrow T$. We obtain:

- $(S, \sigma, \sigma, \mathit{false}) \models T$, since $\Vdash_{\mathit{fol}} \mathit{Res}(\mathit{false}) \Rightarrow T$ and $(S, \sigma, \sigma, \mathit{false}) \models \mathit{Res}(\mathit{false})$.
- $\Sigma \succeq \Sigma$, trivially.
- $\Sigma \models \sigma$, by hypothesis.
- $\Sigma \models \mathit{false} : (\mathit{Bool}, S)$, by the definitions.

Conditional

$$\frac{S(x) = \mathit{false} \qquad \sigma, S \vdash a_1 \rightsquigarrow v, \sigma'}{\sigma, S \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 \rightsquigarrow v, \sigma'}$$

$$\frac{S(x) = \mathit{true} \qquad \sigma, S \vdash a_0 \rightsquigarrow v, \sigma'}{\sigma, S \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 \rightsquigarrow v, \sigma'}$$

We argue the case for *false*. Let a be $\mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1$. Suppose $\Sigma \models \sigma$, $\Sigma \models S : E$, and $E \Vdash a : A :: T$. Since $E \Vdash a : A :: T$, we must have $E \Vdash x : \mathit{Bool} :: \mathit{Res}(x)$ and there must exist A_1, A_1', T_1, and T_1' such that $E \Vdash a_1 : A_1 :: T_1$, $A_1[\mathit{false}/x] = A_1'[\mathit{false}/x]$, $T_1[\mathit{false}/x] = T_1'[\mathit{false}/x]$, $\Vdash A_1' <: A$, and $\Vdash_{\mathit{fol}} T_1' \Rightarrow T$. By induction hypothesis, $(S, \sigma, \sigma', v) \models T_1$ and there exists Σ' such that $\Sigma' \succeq \Sigma$, $\Sigma' \models \sigma'$, and $\Sigma' \models v : (A_1, S)$. We obtain:

- $(S, \sigma, \sigma', v) \models T$: Since $(S, \sigma, \sigma', v) \models T_1$ and $S(x) = \mathit{false}$, we have $(S, \sigma, \sigma', v) \models T_1[\mathit{false}/x]$. But $T_1[\mathit{false}/x] = T_1'[\mathit{false}/x]$, so we have $(S, \sigma, \sigma', v) \models T_1'[\mathit{false}/x]$. Since $S(x) = \mathit{false}$, we have $(S, \sigma, \sigma', v) \models T_1'$. Finally, $\Vdash_{\mathit{fol}} T_1' \Rightarrow T$ yields $(S, \sigma, \sigma', v) \models T$.
- $\Sigma' \succeq \Sigma$.
- $\Sigma' \models \sigma'$.
- $\Sigma' \models v : (A, S)$: Since $\Sigma' \models v : (A_1, S)$, there exist A' and S' such that $\Sigma' \models_0 v : (A', S')$ and $A'S' <: A_1 S$. Since $S(x) = \mathit{false}$ and $A_1[\mathit{false}/x] = A_1'[\mathit{false}/x]$, we obtain $A_1 S = A_1' S$. Since $\Vdash A_1' <: A$, the substitution lemma yields $A_1' S <: AS$. We obtain $A'S' <: A_1 S = A_1' S <: AS$, and hence $\Sigma' \models v : (A, S)$.

Let

$$\frac{\sigma, S \vdash a \rightsquigarrow v, \sigma' \qquad \sigma', S.(x \mapsto v) \vdash b \rightsquigarrow v', \sigma''}{\sigma, S \vdash \mathit{let}\ x = a\ \mathit{in}\ b \rightsquigarrow v', \sigma''}$$

Let c be $\mathit{let}\ x = a\ \mathit{in}\ b$. Suppose $\Sigma \models \sigma$, $\Sigma \models S : E$, and $E \Vdash c : B :: T$. Since $E \Vdash c : B :: T$, we must have $E \Vdash a : A :: R$ and

$E, x{:}\, A \Vdash b : B' :: T'$ for some R, B', and T' such that $\Vdash B' <: B$ and $\Vdash_{fol} R[\check{\sigma}/\acute{\sigma}, \check{alloc}/\acute{alloc}, x/r] \wedge T'[\check{\sigma}/\grave{\sigma}, \check{alloc}/\grave{alloc}] \Rightarrow T$; in addition, $E \Vdash B'$ and $E \Vdash T$ is a transition relation, so x does not occur free in either B' or T, and $\check{\sigma}$ and $\check{alloc}$ do not occur in T. By induction hypothesis, $(S, \sigma, \sigma', v) \models R$ and there exists Σ' such that $\Sigma' \succeq \Sigma$, $\Sigma' \models \sigma'$, and $\Sigma' \models v : (A, S)$. Since $\Sigma \models S : E$, the extension lemma yields $\Sigma' \models S : E$. Therefore, $\Sigma' \models S.(x \mapsto v) : (E, x{:}\, A)$. (Note that x cannot appear in E because $E, x{:}\, A \Vdash b : B' :: T$; hence x is not in the domain of S either.) By induction hypothesis, $(S.(x \mapsto v), \sigma', \sigma'', v') \models T'$ and there exists Σ'' such that $\Sigma'' \succeq \Sigma'$, $\Sigma'' \models \sigma''$, and $\Sigma'' \models v' : (B', S.(x \mapsto v))$. We obtain:

- $(S, \sigma, \sigma'', v') \models T$: Since $(S, \sigma, \sigma', v) \models R$ and $(S.(x \mapsto v), \sigma', \sigma'', v') \models T'$, the two models associated with $(S.(x \mapsto v), \sigma, \sigma', v')$ and $(S.(x \mapsto v), \sigma', \sigma'', v')$ can be extended to a model for $R[\check{\sigma}/\acute{\sigma}, \check{alloc}/\acute{alloc}, x/r] \wedge T'[\check{\sigma}/\grave{\sigma}, \check{alloc}/\grave{alloc}]$. (The interpretation of $\check{\sigma}$ and $\check{alloc}$ in this model is easily determined from σ'.) Since $\Vdash_{fol} R[\check{\sigma}/\acute{\sigma}, \check{alloc}/\acute{alloc}, x/r] \wedge T'[\check{\sigma}/\grave{\sigma}, \check{alloc}/\grave{alloc}] \Rightarrow T$, this is also a model for T. Since $\check{\sigma}$ and $\check{alloc}$ do not occur in T and x does not occur free in T, we conclude that $(S, \sigma, \sigma'', v') \models T$.

- $\Sigma'' \succeq \Sigma$, by transitivity.

- $\Sigma'' \models \sigma''$.

- $\Sigma'' \models v' : (B, S)$: Since $\Sigma'' \models v' : (B', S.(x \mapsto v))$, there exist B'' and S'' such that $\Sigma'' \models_0 v' : (B'', S'')$ and $B''S'' <: B'(S.(x \mapsto v))$. Since x does not occur free in B', we have $B'(S.(x \mapsto v)) = B'S$. Since $\Vdash B' <: B$, the substitution lemma yields $B'S <: BS$. Therefore, $B''S'' <: BS$, so $\Sigma'' \models v' : (B, S)$.

Object construction

$$\frac{S(x_i) = v_i \ {}^{i \in 1..n} \qquad h \notin dom(\sigma) \qquad h \in \mathcal{H} \\ \sigma' = \sigma.(h \mapsto (\mathrm{f}_i \mapsto v_i \ {}^{i \in 1..n},\ \mathrm{m}_j \mapsto \langle \varsigma(y_j)b_j, S \rangle \ {}^{j \in 1..m}))}{\sigma, S \vdash [\mathrm{f}_i = x_i \ {}^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j)b_j \ {}^{j \in 1..m}] \rightsquigarrow h, \sigma'}$$

Let c be $[\mathrm{f}_i = x_i \ {}^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j)b_j \ {}^{j \in 1..m}]$. Suppose $\Sigma \models \sigma$, $\Sigma \models S : E$, and $E \Vdash c : A :: T$. Since $E \Vdash c : A :: T$, there exists A' of the form $[\mathrm{f}_i{:}\, A_i \ {}^{i \in 1..n},\ \mathrm{m}_j{:}\, \varsigma(y_j)B_j :: T_j \ {}^{j \in 1..m}]$ such that $\Vdash A' <: A$, $E \Vdash x_i : A_i :: Res(x_i)$ for $i \in i..n$, and $E, y_j{:}\, A' \Vdash b_j : B_j :: T_j$ for $j \in i..m$. In addition, let T' be:

$$\begin{array}{l}\neg \grave{alloc}(r) \wedge \acute{alloc}(r) \wedge \\ (\forall z \,.\, z \neq r \Rightarrow (\grave{alloc}(z) \equiv \acute{alloc}(z)) \,) \wedge \\ \acute{\sigma}(r, \mathrm{f}_1) = x_1 \wedge \cdots \wedge \acute{\sigma}(r, \mathrm{f}_n) = x_n \wedge \\ (\forall z, w \,.\, z \neq r \Rightarrow \grave{\sigma}(z, w) = \acute{\sigma}(z, w) \,)\end{array}$$

It must be that $\Vdash_{fol} T' \Rightarrow T$. Let Σ' be $\Sigma.(h \mapsto (A', S))$. We obtain:

- $(S, \sigma, \sigma', h) \models T$, since $(S, \sigma, \sigma', h) \models T'$.
- $\Sigma' \succeq \Sigma$, since Σ and σ have the same domain and this domain does not include h.
- $\Sigma' \models \sigma'$: First, Σ' and σ' have the same domain, namely the domain of Σ and σ extended with h. Since $\Sigma \models \sigma$, we need to check conditions only for h; these conditions are determined by $\Sigma'(h) = (A', S)$.
  - For $i \in 1..n$, $\sigma'(h)(\mathrm{f}_i)$ is defined, and equals v_i. Since $E \Vdash x : A_i :: Res(x_i)$, it must be that E contains $x_i\colon A'_i$ for some A'_i such that $\Vdash A'_i <: A_i$. Since $\Sigma \models S : E$ and $S(x_i) = v_i$, we must have $\Sigma \models v_i : (A'_i, S'_i)$ for some prefix S'_i of S, and hence $\Sigma \models_0 v_i : (A''_i, S''_i)$ for some (A''_i, S''_i) such that $A''_i S''_i <: A'_i S'_i$. Therefore, $\Sigma' \models_0 v_i : (A''_i, S''_i)$ (because $\Sigma' \succeq \Sigma$). Since $\Vdash A'_i <: A_i$, the substitution lemma yields $A'_i S <: A_i S$. In addition $A'_i S'_i = A'_i S$. By transitivity, we obtain $A''_i S''_i <: A_i S$. It follows that $\Sigma' \models v_i : (A_i, S)$.
  - For $j \in 1..m$, $\sigma'(v)(\mathrm{m}_j)$ is of the form $\langle \varsigma(y_j)b_j, S\rangle$, and $E, y_j\colon A' \Vdash b_j : B_j :: T_j$. In addition, E is such that $\Sigma \models S : E$, and hence $\Sigma' \models S : E$ by the extension lemma.
- $\Sigma' \models h : (A, S)$, since $\Sigma'(h) = (A', S)$ and $A'S <: AS$ (because $\Vdash A' <: A$ and by the substitution lemma).

Field selection

$$\frac{S(x) = h \qquad h \in \mathcal{H} \qquad \sigma(h)(\mathrm{f}) = v}{\sigma, S \vdash x.\mathrm{f} \rightsquigarrow v, \sigma}$$

Suppose $\Sigma \models \sigma$, $\Sigma \models S : E$, and $E \Vdash x.\mathrm{f} : A :: T$. Since $E \Vdash x.\mathrm{f} : A :: T$, there exist B and A' such that B has the form $[\ldots \mathrm{f}\colon A' \ldots]$, $\Vdash A' <: A$, and the environment E contains $x\colon B$. In addition, $\Vdash_{fol} Res(\grave{\sigma}(x, \mathrm{f})) \Rightarrow T$. We obtain:

- $(S, \sigma, \sigma, v) \models T$, since $v = \sigma(S(x))(\mathrm{f})$ so $(S, \sigma, \sigma, v) \models \mathit{Res}(\grave{\sigma}(x, \mathrm{f}))$.
- $\Sigma \succeq \Sigma$, trivially.
- $\Sigma \models \sigma$, trivially.
- $\Sigma \models v : (A, S)$: Since $\Sigma \models S : E$ and $S(x) = h$, we must have $\Sigma \models h : (B, S')$ for some prefix S' of S, and hence $\Sigma(h) = (B'', S'')$ for some (B'', S'') such that $B''S'' <: BS'$. Therefore, B'' has the form $[\ldots \mathrm{f}{:}\, A'' \ldots]$ with $A''S'' = A'S'$. Moreover, $A'S' = A'S$ since A' is a subexpression of B, (B, S') is a specification closure, and S' is a prefix of S. Since $\Sigma \models \sigma$, $\sigma(h)(f) = v$, and $\Sigma(h) = ([\ldots \mathrm{f}{:}\, A'' \ldots], S'')$, we have $\Sigma \models v : (A'', S'')$, and hence $\Sigma \models_0 v : (A''', S''')$ for some (A''', S''') such that $A'''S''' <: A''S''$. Since $\Vdash A' <: A$, the substitution lemma yields $A'S <: AS$. By transitivity, we derive $A'''S''' <: AS$. Therefore, $\Sigma \models v : (A, S)$.

Method invocation

$$\frac{\begin{array}{lll} S(x) = h & h \in \mathcal{H} & \sigma(h)(\mathrm{m}) = \langle \varsigma(y)b, S' \rangle \\ \sigma, S'.(y \mapsto h) \vdash b \rightsquigarrow v, \sigma' & & \end{array}}{\sigma, S \vdash x.\mathrm{m} \rightsquigarrow v, \sigma'}$$

Suppose $\Sigma \models \sigma$, $\Sigma \models S : E$, and $E \Vdash x.\mathrm{m} : A :: T$. Since $E \Vdash x.\mathrm{m} : A :: T$, there exist B, A', T', and T^+ such that B has the form $[\ldots \mathrm{m}{:}\, \varsigma(y)A' :: T' \ldots]$, $\Vdash A'[x/y] <: A$, $\Vdash_{fol} T' \Rightarrow T^+$, and $\Vdash_{fol} T^+[x/y] \Rightarrow T$, and the environment E contains $x{:}\, B$. Since $\Sigma \models S : E$ and $S(x) = h$, we must have $\Sigma \models h : (B, S^+)$ for some prefix S^+ of S, and hence $\Sigma(h) = (B'', S'')$ for some (B'', S'') such that $B''S'' <: BS^+$. We assume (without loss of generality) that y is not in the domain of S'' or S. Therefore, B'' has the form $[\ldots \mathrm{m}{:}\, \varsigma(y)A'' :: T'' \ldots]$ with $A''S'' <: A'S^+$ and $T''S'' \Rightarrow T'S^+$. Since $\Sigma \models \sigma$, $\sigma(h)(m)$ contains the stack S', and $\Sigma(h) = ([\ldots \mathrm{m}{:}\, \varsigma(y)A'' :: T'' \ldots], S'')$, we obtain that $S'' = S'$. In addition, there exists E' such that $\Sigma \models S' : E'$ and $E', y{:}\, B'' \Vdash b : A'' :: T''$. Since $\Sigma(h) = (B'', S'')$ and $S'' = S'$, we obtain that $\Sigma \models S'.(y \mapsto h) : (E', y{:}\, B'')$. (Note that y cannot appear in E' because $E', y{:}\, B'' \Vdash b : A'' :: T''$.) By induction hypothesis, $(S'.(y \mapsto h), \sigma, \sigma', v) \models T''$ and there exists Σ' such that $\Sigma' \succeq \Sigma$, $\Sigma' \models \sigma'$, and $\Sigma' \models v : (A'', S'.(y \mapsto h))$, and hence there exists (C, U) such that $\Sigma' \models_0 v : (C, U)$ and $CU <: A''(S'.(y \mapsto h))$. We obtain:

- $(S, \sigma, \sigma', v) \models T$: We have $(S'.(y \mapsto h), \sigma, \sigma', v) \models T''$, so $(S''.(y \mapsto h), \sigma, \sigma', v) \models T''$ since $S'' = S'$. From $T''S'' \Rightarrow T'S^+$ we deduce

$(S^+.(y \mapsto h), \sigma, \sigma', v) \models T'$. The free variables of T' other than y are included in the domain of S^+, since (B, S^+) is a specification closure and $\varsigma(y)A' :: T'$ is a subexpression of B. In addition, S^+ is a prefix of S. Therefore, $(S^+.(y \mapsto h), \sigma, \sigma', v) \models T'$ implies $(S.(y \mapsto h), \sigma, \sigma', v) \models T'$. We obtain $(S.(y \mapsto h), \sigma, \sigma', v) \models T^+$ since $\Vdash_{fol} T' \Rightarrow T^+$, so $(S, \sigma, \sigma', v) \models T^+[x/y]$ by substitution and since $S(x) = h$, so $(S, \sigma, \sigma', v) \models T$ since $\Vdash_{fol} T^+[x/y] \Rightarrow T$.

- $\Sigma' \succeq \Sigma$.
- $\Sigma' \models \sigma'$.
- $\Sigma' \models v : (A, S)$: We already have $\Sigma' \models_0 v : (C, U)$. In addition, $CU <: A''(S'.(y \mapsto h)) = A''(S''.(y \mapsto h)) <: A'(S^+.(y \mapsto h)) = A'(S.(y \mapsto h)) = A'[x/y]S <: AS$. (The equality $A''(S'.(y \mapsto h)) = A''(S''.(y \mapsto h))$ follows from $S'' = S'$. The relation $A''(S''.(y \mapsto h)) <: A'(S^+.(y \mapsto h))$ is obtained from $A''S'' <: A'S^+$ by substitution. The equality $A'(S^+.(y \mapsto h)) = A'(S.(y \mapsto h))$ holds because $\varsigma(y)A' :: T'$ is a subexpression of B, (B, S^+) is a specification closure, and S^+ is a prefix of S. The equality $A'(S.(y \mapsto h)) = A'[x/y]S$ holds because $S(x) = h$. The relation $A'[x/y]S <: AS$ follows from $\Vdash A'[x/y] <: A$ by the substitution lemma.)

Field update

$$\frac{\begin{array}{lll} S(x) = h & h \in \mathcal{H} & \sigma(h)(\mathrm{f}) \text{ is defined} \\ S(y) = v & \multicolumn{2}{l}{\sigma' = \sigma.(h \mapsto \sigma(h).(\mathrm{f} \mapsto v))} \end{array}}{\sigma, S \vdash x.\mathrm{f} := y \rightsquigarrow h, \sigma'}$$

Suppose $\Sigma \models \sigma$, $\Sigma \models S : E$, and $E \Vdash x.\mathrm{f} := y : A :: T$. Since $E \Vdash x.\mathrm{f} := y : A :: T$, there exist A' and C such that A' has the form $[\ldots \mathrm{f}{:}\, C \ldots]$, $\Vdash A' <: A$, and the environment E contains $x{:}\, A'$; and there exists C' such that $\Vdash C' <: C$ and the environment E contains $y{:}\, C'$. In addition, let T' be:

$$\begin{array}{l} r = x \wedge \acute{\sigma}(x, \mathrm{f}_k) = y \wedge \\ (\,\forall z, w\,.\, \neg(z = x \wedge w = \mathrm{f}_k) \Rightarrow \grave{\sigma}(z, w) = \acute{\sigma}(z, w)\,) \wedge \\ (\,\forall z\,.\, \grave{alloc}(z) \equiv \acute{alloc}(z)\,) \end{array}$$

It must be that $\Vdash_{fol} T' \Rightarrow T$. We obtain:

- $(S, \sigma, \sigma', h) \models T$ since $(S, \sigma, \sigma', h) \models T'$.
- $\Sigma \succeq \Sigma$, trivially.

- $\Sigma \models \sigma'$: Since $\Sigma \models \sigma$, we need to check conditions only for $\sigma'(h)(\mathrm{f})$. Since $\Sigma \models S : E$ and $S(x) = h$, we must have $\Sigma \models h : (A', S')$ for some prefix S' of S, and hence $\Sigma(h) = (A'', S'')$ for some (A'', S'') such that $A''S'' <: A'S'$. Therefore, A'' has the form $[\ldots \mathrm{f}{:}\, C'' \ldots]$ with $C''S'' = CS'$. Since $\Sigma \models S : E$ and $S(y) = v$, we must have $\Sigma \models v : (C', U)$ for some prefix U of S, and hence $\Sigma \models_0 v : (B', U')$ for some (B', U') such that $B'U' <: C'U$. Because $\Sigma(h)$ has the form $([\ldots \mathrm{f}{:}\, C'' \ldots], S'')$, we need that $\Sigma \models v : (C'', S'')$. We obtain this from $\Sigma \models_0 v : (B', U')$ and $B'U' <: C'U = C'S <: CS = CS' = C''S''$. (The relation $C'S <: CS$ follows from $\Vdash C' <: C$ by the substitution lemma. The equality $CS = CS'$ holds because C is a subexpression of A', (A', S') is a specification closure, and S' is a prefix of S.)

- $\Sigma \models h : (A, S)$: We have $A''S'' <: A'S'$; moreover, $A'S' = A'S$ since (A', S') is a specification closure and S' is a prefix of S. Since $\Vdash A' <: A$, the substitution lemma yields $A'S <: AS$. By transitivity, we derive $A''S'' <: AS$. Therefore, $\Sigma(h) = (A'', S'')$ yields $\Sigma \models h : (A, S)$.

□

References

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

[Bar77] J. Barwise. An introduction to first-order logic. In J. Barwise, editor, *The Handbook of Mathematical Logic*, Studies in Logic and Foundations of Mathematics, pages 5–46. North Holland, 1977.

[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.

[dB90] F. S. de Boer. A proof system for the parallel object-oriented laguage POOL. In M. S. Paterson, editor, *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 572–585. Springer-Verlag, 1990.

[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.

[Lei98] K. R. M. Leino. Recursive object types in a logic of object-oriented programs. In C. Hankin, editor, *Programming Languages and Systems: 7th European Symposium on Programming, ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 170–184. Springer-Verlag, April 1998.

[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 O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 78–97. Springer-Verlag, 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.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997. Available as `http://wwweickel.informatik.tu-muenchen.de/persons/poetzsch/habil.ps.gz`.

[PHM98] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W.-P. de Roever, editors, *Programming Concepts and Methods (PROCOMET'98)*, IFIP, pages 404–423. Chapman & Hall, June 1998.

[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, 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.