Nordic Journal of Computing 5(1998), 330–360.

RECURSIVE OBJECT TYPES IN A LOGIC OF OBJECT-ORIENTED PROGRAMS

K. RUSTAN M. LEINO
Compaq Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
rustan@pa.dec.com

Abstract. This paper formalizes a small object-oriented programming notation. The notation features imperative commands where objects can be shared (aliased), and is rich enough to allow subtypes and recursive object types. The syntax, type checking rules, axiomatic semantics, and operational semantics of the notation are given. A soundness theorem showing the consistency between the axiomatic and operational semantics is stated and proved. A simple corollary of the soundness theorem demonstrates the soundness of the type system. Because of the way types, fields, and methods are declared, no extra effort is required to handle recursive object types.

CR Classification: D., D.3, D.3.1

Key words: programming logic, object-oriented programs, recursive object types, program correctness, axiomatic semantics, soundness

1. Introduction

It is well known that C.A.R. Hoare's logic of the basic commands of imperative, procedural languages [Hoare 1969] has been useful in understanding imperative languages. Object-oriented programming languages being all the rage, one is surprised that the literature has not produced a corresponding logic for modern object-oriented programs. The control structures of object-oriented programs are similar to those treated by Hoare, but the data structures of object-oriented programs are more complicated, mainly because objects are (possibly shared) references to data fields.

This paper presents a logic for an object-oriented programming notation. In an early attempt at such a logic, Leavens [1989] gave an axiomatic semantics for an object-oriented language. However, the language he used differs from popular object-oriented languages in that it is functional rather than imperative, so the values of the fields of objects cannot be changed. America and de Boer [1994] have given a logic for the parallel language POOL. This logic applies to imperative programs with object sharing (sometimes called aliasing), but without subtyping and method overriding. In a logic that I will refer to as logic AL, Abadi and I defined an axiomatic semantics for an imperative, object-oriented language with object sharing [Abadi

Received July 1998; revised October 1998; accepted October 1998.

and Leino 1997], but it does not permit recursive object types. Poetzsch-Heffter and Müller [1998] have defined (but not proved sound) a Hoare-style logic for object-oriented programs that remove many of the previous limitations. However, instead of following the standard methodological discipline of letting the designer of a method define its specification and then checking that implementations meet the specification, the specification of a method in the Poetzsch-Heffter and Müller logic is derived from the method's known implementations. The present logic deals with imperative features, subtyping, and recursive object types.

The literature has paid much attention to the type systems of object-oriented languages. Such papers tend to define some notion of types, the commands of some language, the type rules and operational semantics for the commands, and a soundness theorem linking the type system with the operational semantics. (Several examples of this are found in Abadi and Cardelli's book on objects [Abadi and Cardelli 1996].) But after all that effort, one still doesn't know how to *reason* about the programs that can be written with the provided commands, since no axiomatic semantics is given. In addition to giving a programming notation and its axiomatic semantics, this paper, like the paper describing logic AL, gives an operational semantics and a soundness theorem that links the operational semantics with the axiomatic semantics. The soundness theorem directly implies the soundness of the type system.

A complication with type systems for object-oriented languages is that types can be *recursive*, that is, an object type T may contain a field of type T or a method whose return type is T. The literature commonly treats recursive data types by introducing some sort of fix-point operator into the type system, good examples of which are a paper by Amadio and Cardelli on recursive types and subtypes [Amadio and Cardelli 1993] and the book by Abadi and Cardelli. By treating types in a dramatically different way, the present logic supports recursive object types without the need for any special mechanism like fix-points. The inclusion of recursive object types is one main advantage of the present logic over logic AL, which does not allow them. (The other main advantage over logic AL is that the present logic can be used with any first-order theory.) Because the given soundness theorem implies the soundness of the type system, the present work contributes also to the world of type systems.

In contrast to the paper by Amadio and Cardelli, which considers unrestricted recursive types, the type system in the present paper uses a restriction along the lines of name matching. In particular, types are simply identifiers, and the subtype relation is simply a given partial order among those identifiers. This is much like the classes in Java [Gosling *et al.* 1996] or the branded object types in Modula-3 [Nelson 1991]. But in contrast to languages like Java or Modula-3, fields and methods are declared separately from types in the language considered in this paper. (This is also done in Cecil [Chambers 1997] and Ecstatic [Leino 1997].) Not only does this simplify the treatment without loss of applicability to languages like Java and

Modula-3, but it also makes explicit the separation of concerns. For example, as the logic shows, having to know all the fields of a particular object type is necessary only for the allocation of a new object.

Furthermore, when a field or method is declared at some type T, each subtype of T automatically acquires, or *inherits*, that field or method. Consequently, one gets behavioral subtyping for free, something that can also be achieved by the inheritance discipline considered by Dhara and Leavens [1997].

The rest of this paper is organized as follows. Section 2 relates the present logic to some work that has influenced it. Section 3 describes the declarations that can be used in program environments, and Section 4 describes the commands: their syntax, axiomatic semantics, and operational semantics. Section 5 discusses an example program. Then, Section 6 states the soundness theorem, whose proof is given in the appendix. Section 7 discusses some limitations of the logic, and the paper concludes with a brief summary.

2. Sources of influence

My work with Abadi has inculcated the present logic with its style and machinery. The present logic also draws from other sources with which I am quite familiar: my thesis [Leino 1995], my work on an object logic with Nelson [Leino and Nelson 1995], and the Ecstatic language [Leino 1997]. This section compares the features of these sources of influence with the features of the present logic.

My thesis includes a translation of common object-oriented language constructs into Dijkstra's *guarded commands*, an imperative language whose well-known axiomatic semantics is given in terms of *weakest preconditions* [Dijkstra 1976]. My attempt at an object logic with Nelson is also based on guarded commands, and Ecstatic is a richer object-oriented programming language defined directly in terms of weakest preconditions. Types, fields, and methods in these three sources are declared in roughly the same way as in the present logic. While these sources do provide a way to reason about object-oriented programs, they take for granted the existence of an operational semantics that implements the axiomatic semantics. The present paper includes an operational semantics for the given commands, and establishes the correctness of the operational semantics with respect to the axiomatic semantics by proving a soundness theorem.

Like logic AL, the present logic has few and simple commands. Each command in logic AL operates on an object store and produces a value placed in a special register called r. In the present logic, commands are allowed to refer to the initial value of that register, which simplifies many of the rules. Another difference is that the present logic splits logic AL's *let* command into two commands: sequential composition and binding. The separation works well because the initial value of register r can be used. Perhaps surprisingly, another consequence of using the initial value of r is that the present logic manages fine without Abadi and Cardelli's ς binder that appears in logic AL to bind a method's self parameter.

3. Environments

This section starts defining the logic by describing program environments and the declarations that a program environment can contain.

A *program environment* is a list of declarations. A declaration introduces a type, a field, or a method.

An identifier is said to be declared in an environment if it is introduced by a type, field, or method declaration in the environment, or if it is one of the built-in types. I write $x \notin E$ to denote that identifier x is not declared in environment E.

The judgement $E \vdash \diamond$ says that E is a well-formed environment. The empty list, written $\emptyset$, is a valid environment.

EMPTY ENVIRONMENT

$$\frac{}{\emptyset \vdash \diamond}$$

The next three subsections describe types, fields, and methods, and give the remaining rules for well-formed environments.

3.1 Types

A *type* is an identifier. There are two built-in types, *Boolean* and *Object*. (Other types, like integers, can easily be added, but I omit them for brevity.) Types other than *Boolean* are called *object types*. A new object type is introduced by a *subtyping pair*, which has the form $T{<:}U$, where T is an identifier that names the new type, and U is an object type. Like in Java, *Object* denotes the root of the class hierarchy. The analogue of a subtyping pair $T{<:}U$ in Java is a class T declared as a subclass of a class U:

$$\mathit{class}\ T\ \mathit{extends}\ U\ \{\ \dots\ \}$$

A type is said to be declared in an environment if it is *Boolean*, *Object*, or if it occurs as the first component of a subtyping pair. To express this formally, the judgement $E \vdash_{type} T$ says that T is a type in environment E, and the judgement $E \vdash_{obj} T$ says that T is an object type in E. The rules for these judgements are as follows. Here and throughout this paper, I use T and U, possibly subscripted, to denote types.

TYPES IN ENVIRONMENTS

$$\frac{E \vdash_{obj} U \qquad T \notin E}{(E, T{<:}U) \vdash \diamond}$$

DECLARED TYPES $\quad (\vdash_{type}, \vdash_{obj})$

$$\frac{E \vdash \diamond}{E \vdash_{obj} \mathit{Object}} \qquad \frac{(E, T{<:}U, E') \vdash \diamond}{(E, T{<:}U, E') \vdash_{obj} T}$$

$$\frac{E \vdash \diamond}{E \vdash_{type} Boolean} \qquad \frac{E \vdash_{obj} T}{E \vdash_{type} T}$$

The "Types in environments" rule requires that all types in a program have distinct names. Programming languages often allow types introduced in different scopes to have the same declared names. These declared names are, or can be viewed as being, contextual syntactic shorthands for fully qualified names that are distinct. Since it is possible for a programmer to adhere to the logic's requirement of distinct names for types by supplying names that are already fully qualified, the requirement simplifies the logic without losing theoretical expressiveness.

The reflexive, transitive closure of the subtyping pairs forms a partial order called the *subtyping order*. The judgement $E \vdash T <: U$ says that T and U are types in E that are ordered by the subtyping order. Type T is then said to be a *subtype* of U. The rules are:

SUBTYPING ORDER $(\vdash <:)$

$$\frac{E \vdash_{type} T}{E \vdash T <: T} \qquad \frac{(E, T{<:}U, E') \vdash \diamond}{(E, T{<:}U, E') \vdash T <: U} \qquad \frac{E \vdash T_0 <: T_1 \qquad E \vdash T_1 <: T_2}{E \vdash T_0 <: T_2}$$

3.2 Fields

A field is a map from an object type to another type. A field f is introduced by a *field triple*, written $\mathsf{f}: T \to U$, where f is the identifier that names the field, T is an object type called the *index type* of f, and U is a type called the *range type* of f. The analogue of a field triple $\mathsf{f}: T \to U$ in Java is an instance variable f of type U declared in a class T:

$$class\ T\ \{\ \ldots\ U\ \mathsf{f};\ \ldots\ \}$$

An environment can contain field triples. A field f is said to be declared in an environment E if it occurs in some field triple $\mathsf{f}: T \to U$ in E. This is expressed by the judgement $E \vdash_{field} \mathsf{f}: T \to U$. The rules for these judgements are as follows. Here and throughout, I use f, possibly subscripted, to denote field names.

FIELDS IN ENVIRONMENTS

$$\frac{E \vdash_{obj} T \qquad E \vdash_{type} U \qquad \mathsf{f} \notin E}{(E, \mathsf{f}: T \to U) \vdash \diamond}$$

DECLARED FIELDS $(\vdash_{field})$

$$\frac{(E, \mathsf{f}: T \to U, E') \vdash \diamond}{(E, \mathsf{f}: T \to U, E') \vdash_{field} \mathsf{f}: T \to U}$$

The "Fields in environments" rule requires that all fields in a program have distinct names. Like the requirement of distinct names for types, this requirement simplifies the logic without losing theoretical expressiveness.

For a type T_0 declared in an environment E, the *set of fields of* T_0 in E, written $\mathit{Fields}(T_0, E)$, is the set of all field triples $\mathsf{f}: T \to U$ such that $E \vdash_{\mathit{field}} \mathsf{f}: T \to U$ and $E \vdash T_0 <: T$.

3.3 Methods

A *method quadruple* has the form $\mathsf{m}: T \to U : R$, where m is an identifier denoting a *method*, T is an object type, U is a type, and R is a *relation*. The analogue of a method quadruple $\mathsf{m}: T \to U : R$ in Java is a method m with return type U declared in a class T and given a specification R:

$$\mathit{class}\ T\ \{\ \ldots\ U\ \mathsf{m}()\ \{\ldots\}\ \ldots\ \}$$

Note that the Java language does not have a place to write down the specification of a method. In the present language, the declaration of a method includes a specification, which specifies the effect of the method as a relation on the pre- and post-state of each method invocation. Note also that methods take no parameters (other than the object on which the method is invoked, an object commonly referred to as *self*). This simplifies the logic without losing theoretical expressiveness, since parameters can be passed through fields.

An environment can contain method quadruples. A method m is said to be declared in an environment E if it occurs in some method quadruple $\mathsf{m}: T \to U : R$ in E. This is expressed by the judgement $E \vdash_{\mathit{method}} \mathsf{m}: T \to U : R$. Formally, the rules are as follows. I use m, possibly subscripted, to denote methods.

METHODS IN ENVIRONMENTS

$$\frac{E \vdash_{\mathit{obj}} T \qquad E \vdash_{\mathit{type}} U \qquad E, \emptyset \vdash_{\mathit{rel}} R \qquad \mathsf{m} \notin E}{(E, \mathsf{m}: T \to U : R) \vdash \diamond}$$

DECLARED METHODS $\quad (\vdash_{\mathit{method}})$

$$\frac{(E, \mathsf{m}: T \to U : R, E') \vdash \diamond}{(E, \mathsf{m}: T \to U : R, E') \vdash_{\mathit{method}} \mathsf{m}: T \to U : R}$$

The judgement $E, \emptyset \vdash_{\mathit{rel}} R$, which will be described in more detail in Section 4.2, essentially says that R is a relation that may mention fields declared in E but doesn't mention any local program variables.

The "Methods in environments" rule requires that all methods in a program have distinct names. Like the requirements of distinct names for types and fields, this requirement simplifies the logic without losing theoretical expressiveness.

For a type T_0 declared in an environment E, the *set of methods of* T_0 in E, written $\mathit{Methods}(T_0, E)$, is the set of all method quadruples $\mathsf{m}: T \to U : R$ such that $E \vdash_{\mathit{method}} \mathsf{m}: T \to U : R$ and $E \vdash T_0 <: T$.

3.4 Relations

Methods are specified using *relations.* A relation is an untyped first-order predicate on a pre-state and a post-state. In order to make relations expressive, the present logic can be used with an *underlying logic*, which provides a set of function symbols and a set of first-order axioms about those function symbols. The example in Section 5 shows how an underlying logic may be used.

Syntactically, relations are made up only of:

- the constants *false*, *true*, and *nil*;
- constants for field names and the special field alloc;
- the special variables $\grave{r}$, $\acute{r}$, $\grave{\sigma}$, $\acute{\sigma}$ (explained below);
- other variables (I will write v to denote a typical variable);
- equality between terms;
- applications of the functions *select* and *store* (explained below);
- applications of the functions of the underlying logic;
- the usual logical connectives $\neg$, $\wedge$, and $\forall$.

The grammars for relations (R) and terms (e) are thus:

$$
\begin{array}{lcl}
R & ::= & e_0 = e_1 \ \mid\ \neg R \ \mid\ R_0 \wedge R_1 \ \mid\ \langle \forall x :: R \rangle \\
e & ::= & \mathit{false} \ \mid\ \mathit{true} \ \mid\ \mathit{nil} \ \mid\ \mathsf{f} \\
 & \mid & \grave{r} \ \mid\ \acute{r} \ \mid\ \grave{\sigma} \ \mid\ \acute{\sigma} \ \mid\ v \\
 & \mid & \mathit{select}(e_0, e_1, e_2) \ \mid\ \mathit{store}(e_0, e_1, e_2, e_3) \\
 & \mid & \mathit{pa}(e_0, \ldots, e_{ka-1}) \ \mid\ \cdots \ \mid\ \mathit{pz}(e_0, \ldots, e_{kz-1})
\end{array}
$$

where $pa, \ldots, pz$ denote the function symbols of the underlying logic with arities $ka, \ldots, kz$, respectively. It will be convenient also to allow $\neq$, $\vee$, $\Rightarrow$, $\Leftarrow$, $\equiv$, and $\exists$ as the usual abbreviations of derived operators.

The semantics of a command (program statement) is defined in terms of a relation on a *register* and a *(data) store*, together called a *state*. The variables $\grave{r}$ and $\acute{r}$ denote the register in the pre- and post-states of the command, respectively, and $\grave{\sigma}$ and $\acute{\sigma}$ denote the store in those respective states. The value of a field f of an object e in a store σ is denoted $\mathit{select}(\sigma, e, \mathsf{f})$. The expression $\mathit{store}(\sigma, e_0, \mathsf{f}, e_1)$ represents the store that results from setting the f field of object e_0 in store σ to the value e_1. The relationship between *select* and *store* is defined as follows.

$$
\begin{array}{l}
\langle \forall \sigma, e_0, e_1, \mathsf{f}_0, \mathsf{f}_1, e :: \\
\quad \mathit{select}(\mathit{store}(\sigma, e_0, \mathsf{f}_0, e), e_0, \mathsf{f}_0) = e \ \wedge \\
\quad (e_0 \neq e_1 \vee \mathsf{f}_0 \neq \mathsf{f}_1 \Rightarrow \\
\qquad \mathit{select}(\mathit{store}(\sigma, e_0, \mathsf{f}_0, e), e_1, \mathsf{f}_1) = \mathit{select}(\sigma, e_1, \mathsf{f}_1)) \rangle
\end{array}
\tag{1}
$$

The special field alloc is used to record which objects in the data store have been allocated; the alloc field of an object is *false* until the object is allocated, and is *true* from there on.

As we shall see, the logic allows relations to be rewritten. A rewriting uses the rules of logic and some axioms. In particular, a rewriting may use as axioms the definition of *select* and *store* (Eq. (1)), the distinctness of the boolean values:

$$\mathit{false} \neq \mathit{true} \tag{2}$$

the distinctness of field name constants:

$$\text{all field name constants (including } \mathsf{alloc}\text{) are distinct} \tag{3}$$

and the axioms of the underlying logic.

4. Commands

This section describes commands: their syntax, their axiomatic semantics, and their operational semantics.

4.1 Syntax

A *command* has a form dictated by the following grammar.

$$\begin{array}{lcll}
a & ::= & c & \text{constant} \\
 & | & v & \text{local variable} \\
 & | & a_0 \mathbin{\triangleleft\!\triangleright} a_1 & \text{conditional} \\
 & | & a_0 \ ; \ a_1 & \text{composition} \\
 & | & \mathit{with}\ v{:}T\ \mathit{do}\ a & \text{binding} \\
 & | & [T\!:\ \mathsf{f}_i = c_i \ ^{i \in I},\ \mathsf{m}_j = a_j \ ^{j \in J}] & \text{object construction} \\
 & | & \mathsf{f} & \text{field selection} \\
 & | & \mathsf{f} := v & \text{field update} \\
 & | & \mathsf{m} & \text{method invocation} \\
c & ::= & \mathit{false} \ | \ \mathit{true} \ | \ \mathit{nil} &
\end{array}$$

Informally, the semantics of the language is as follows. (Recall from the previous section that commands operate on a register and a store.)

- The constants *false*, *true*, and *nil* evaluate to themselves. That is, they have the effect of setting the register to themselves.
- A local variable is an identifier introduced via a binding command. Every local variable is immutable: once bound (using *with*, see below), the value of a local variable cannot be changed. A local variable evaluates to its value.
- The conditional command evaluates a_0 if the register is initially *false*, and evaluates a_1 if the register is initially *true*. Note that the guard of the conditional is not shown explicitly in the command; rather, the initial value of the register is used as the guard.

- The sequential composition of a_0 and a_1 first evaluates a_0 and then evaluates a_1. The final values of the register and store in the evaluation of a_0 are used as the initial values of the register and store in the evaluation of a_1. Composition is usually written $a_0\ ;\ a_1$, but to keep the language looking like popular object-oriented languages, I also allow the alternative syntax $a_0\ .\ a_1$ (see examples below).
- The binding command *with* $v{:}T$ *do* a introduces a local variable v for use in a. Its evaluation consists in evaluating a with v bound to the initial value of the register.
- The command $[T\colon\ \mathsf{f}_i = c_i\ ^{i\in I},\ \mathsf{m}_j = a_j\ ^{j\in J}]$ constructs a new object of type T, and sets the register to (a reference to the fields and methods of) the object. The command must list every field f_i from the set of fields of T. The initial value for field f_i is the given constant c_i. The command must also list every method m_j from the set of methods of T. The implementation of method m_j for the new object is given as the command a_j, which receives self as the initial value of the register and returns the method result value as the final value of the register. The command a_j cannot reference local variables other than those it declares.
- A field can be selected (f) and updated ($\mathsf{f} := v$). Both operate on the object referenced by the initial value of the register. Selection sets the register to the f field of the object. Update sets the f field of the object to the value of v, leaving the register unchanged.
- The method invocation m finds the implementation of method m for the object referenced by the initial value of the register, and then proceeds to evaluate that implementation. The evaluation of the implementation begins with the initial register and store values of the invocation, and the invocation ends with the final register and store values of the evaluation of the implementation. Other than the initial and final register values (which encode self and the result value, respectively), a method does not have explicit parameters; instead, parameters can be passed via the fields of the object.

Here are some examples that compare the present commands with programs written in other languages. The Modula-3 program statement

$$\textit{if } b \textit{ then } S \textit{ else } T \textit{ end}$$

is written as the command $b\ ;\ (T \diamond S)$. The Modula-3 object creation expression $\textit{new}(T, \mathsf{f} := \textit{true}).\mathsf{f}$, where T is an object type with one field f and no methods, is written as the command

$$[T : \mathsf{f} = \textit{true}]\ ;\ \mathsf{f}$$

or with the alternative syntax for composition, the command is written

$$[T : \mathsf{f} = \textit{true}].\mathsf{f}$$

The Modula-3 program $x.\mathsf{f} := \mathit{true}$ is written

$$\mathit{true}\ ;\ \mathit{with}\ v{:}\,\mathit{Boolean}\ \mathit{do}\ x.\mathsf{f} := v$$

As an example of object sharing, the command

$$[T{:}\ \mathsf{f} = c]\ ;\ \mathit{with}\ v{:}T\ \mathit{do}\ \mathit{with}\ w{:}T\ \mathit{do}\ (v.\mathsf{f} := y\ ;\ w.\mathsf{f})$$

allocates a new T object whose f field is set to c, creates two references to the object (v and w), updates the object's f field via v, and reads f back via w, returning y.

The following example shows the construction of a T object whose method or computes the disjunction of fields x and y:

$$[T{:}\ \mathsf{x} = \mathit{false},\ \mathsf{y} = \mathit{false},\ \mathsf{or} = \mathit{with}\ \mathit{self}{:}T\ \mathit{do}\ (\mathsf{x}\ ;\ (\mathit{self}.\mathsf{y} \diamond \mathit{true}))]$$

Note that although primitive, the programming notation is expressive enough to admit common object-oriented languages features like object construction, method invocation, and object sharing. The programming notation is kept minimal in order to simplify the associated rules.

4.2 Axiomatic semantics

This subsection gives the axiomatic semantics of the commands. The judgement

$$E, V \vdash a : T \to U : R$$

says that command a in command environment (E, V) can be started in a state where the register contents are of type T, and terminates in a state where the register contents are of type U. The execution of a is such that its pre- and post-states satisfy the relation R. The rules of the axiomatic semantics double as type checking rules, because with a trivial R (such as $\grave{r} = \grave{r}$), the judgement expresses what it means for command a to be well-typed.

Before giving the axiomatic semantics, some other definitions and rules pertaining to constants, local variables, and command environments are in order.

There are three constants: *false*, *true*, and *nil*. The judgement $E \vdash_{const} c{:}T$ expresses that constant c has type T.

TYPE OF CONSTANTS $(\vdash_{const})$

$$\frac{E \vdash \diamond}{E \vdash_{const} \mathit{false}{:}\,\mathit{Boolean}} \qquad \frac{E \vdash \diamond}{E \vdash_{const} \mathit{true}{:}\,\mathit{Boolean}} \qquad \frac{E \vdash_{obj} T}{E \vdash_{const} \mathit{nil}{:}T}$$

A *local variable declaration* has the form $v{:}T$, where v is an identifier denoting a *local variable* and T is a type. A *command environment* is a pair (E, V), where E is a program environment and V is a list of local variable declarations. A local variable v is said to be declared in a command environment (E, V) if it occurs in some local variable declaration $v{:}T$ in V. This is expressed by the judgement $E, V \vdash_{var} v{:}T$. Thus, in a command environment (E, V), E contains declarations of types, fields, and methods, whereas V contains declarations of local variables. This separation allows a simple characterization of a command environment without local variable declarations: $(E, \emptyset)$. We saw this in the "Methods in environments" rule in Section 3.3, and we will see it in the "Object construction" rule below and in Theorems 1, 2, and 3 in Section 6.

The judgement

$$E, V \vdash_{rel} R$$

says that R is a relation whose free variables are fields or local variables declared in (E, V), or are among the special fields and variables alloc, $\grave{r}$, $\acute{r}$, $\grave{\sigma}$, and $\acute{\sigma}$. The obvious formal rules for this judgement are omitted. Thus, the judgement $E, \emptyset \vdash_{rel} R$ used in the hypothesis of the "Methods in environments" rule in Section 3.3 implies that R does not mention local variables.

I write $x \notin (E, V)$ to denote that identifier x is not declared in command environment (E, V). The formal rules of the above are then:

WELL-FORMED COMMAND ENVIRONMENT

$$\frac{E \vdash \diamond}{E, \emptyset \vdash \diamond} \qquad \frac{E, V \vdash \diamond \qquad v \notin (E, V) \qquad E \vdash_{type} T}{E, (V, v{:}T) \vdash \diamond}$$

DECLARED LOCAL VARIABLES $(\vdash_{var})$

$$\frac{E, (V, v{:}T, V') \vdash \diamond}{E, (V, v{:}T, V') \vdash_{var} v{:}T}$$

The "Well-formed command environment" rules require that the names of local variables be distinct from the names of types, fields, and methods declared in the program environment, and from the names of other local variables declared in the same command environment. However, variables introduced by unnested *with* ... *do* commands may have the same names, since such variables never appear in the same command environment (see the "Binding" rule below).

Now for the rules of the axiomatic semantics. There is one rule for each command, and one subsumption rule.

SUBSUMPTION

$$\frac{\begin{array}{c} E, V \vdash a : T_1 \to T_2 : R \\ E \vdash T_0 <: T_1 \qquad E \vdash T_2 <: T_3 \qquad \vdash_{fol} R \Rightarrow R' \qquad E, V \vdash_{rel} R' \end{array}}{E, V \vdash a : T_0 \to T_3 : R'}$$

The judgement $\vdash_{fol} P$ represents provability in first-order logic, under axioms Eq. (1), Eq. (2), and Eq. (3) from Section 3.4 and the axioms of the underlying logic.

Constant

$$\frac{E,V \vdash \diamond \qquad E \vdash_{const} c{:}T \qquad E \vdash_{type} U}{E,V \vdash c : U \to T : \acute{r} = c \wedge \grave{\sigma} = \acute{\sigma}}$$

Local variable

$$\frac{E,V \vdash_{var} v{:}T \qquad E \vdash_{type} U}{E,V \vdash v : U \to T : \acute{r} = v \wedge \grave{\sigma} = \acute{\sigma}}$$

Conditional

$$\frac{\begin{array}{c} E,V \vdash a_0 : \mathit{Boolean} \to T : R_0 \\ E,V \vdash a_1 : \mathit{Boolean} \to T : R_1 \end{array}}{E,V \vdash a_0 \diamond a_1 : \mathit{Boolean} \to T : (\grave{r} = \mathit{false} \Rightarrow R_0) \wedge (\grave{r} = \mathit{true} \Rightarrow R_1)}$$

Composition

$$\frac{\begin{array}{c} E,V \vdash a_0 : T_0 \to T_1 : R_0 \\ E,V \vdash a_1 : T_1 \to T_2 : R_1 \\ \check{r} \text{ and } \check{\sigma} \text{ do not occur free in } R_0 \text{ or } R_1 \end{array}}{E,V \vdash a_0 \ ; \ a_1 : T_0 \to T_2 : \langle \exists\, \check{r}, \check{\sigma} :: \ R_0[\acute{r}, \acute{\sigma} := \check{r}, \check{\sigma}] \wedge R_1[\grave{r}, \grave{\sigma} := \check{r}, \check{\sigma}] \,\rangle}$$

Binding

$$\frac{E,(V,v{:}T) \vdash a : T \to U : R}{E,V \vdash \mathit{with}\ v{:}T\ \mathit{do}\ a : T \to U : R[v := \grave{r}]}$$

Object construction

$$\frac{\begin{array}{l} E,V \vdash \diamond \qquad E \vdash_{type} U \qquad E \vdash_{obj} T \\ \mathsf{f}_i{:}T_i \to U_i \ {}^{i \in I} \text{ are the elements of } \mathit{Fields}(T,E) \\ \mathsf{m}_j{:}T_j \to U_j : R_j \ {}^{j \in J} \text{ are the elements of } \mathit{Methods}(T,E) \\ E \vdash_{const} c_i{:}U_i \ {}^{i \in I} \qquad E,\emptyset \vdash a_j : T \to U_j : R_j \ {}^{j \in J} \end{array}}{\begin{array}{l} E,V \vdash [T{:}\ \mathsf{f}_i = c_i \ {}^{i \in I},\ \mathsf{m}_j = a_j \ {}^{j \in J}] : U \to T : \\ \qquad \acute{r} \neq \mathit{nil} \wedge \mathit{select}(\grave{\sigma}, \acute{r}, \mathsf{alloc}) = \mathit{false} \wedge \\ \qquad \acute{\sigma} = \mathit{store}(\cdots(\mathit{store}(\grave{\sigma}, \acute{r}, \mathsf{alloc}, \mathit{true}), \quad \acute{r}, \mathsf{f}_i, c_i) \ {}^{i \in I} \end{array}}$$

Field selection

$$\frac{E,V \vdash \diamond \qquad E \vdash_{field} \mathsf{f}{:}T \to U}{E,V \vdash \mathsf{f} : T \to U : \grave{r} \neq \mathit{nil} \Rightarrow \acute{r} = \mathit{select}(\grave{\sigma}, \grave{r}, \mathsf{f}) \wedge \grave{\sigma} = \acute{\sigma}}$$

Field update

$$\frac{\begin{array}{ll} E \vdash_{field} \mathsf{f}{:}T_0 \to U_0 & E \vdash T_1 <: T_0 \\ E,V \vdash_{var} v{:}U_1 & E \vdash U_1 <: U_0 \end{array}}{E,V \vdash \mathsf{f} := v : T_1 \to T_1 : \grave{r} \neq \mathit{nil} \Rightarrow \grave{r} = \acute{r} \wedge \acute{\sigma} = \mathit{store}(\grave{\sigma}, \grave{r}, \mathsf{f}, v)}$$

Method invocation

$$\frac{E,V \vdash \diamond \qquad E \vdash_{method} \mathsf{m}{:}T \to U : R}{E,V \vdash \mathsf{m} : T \to U : \grave{r} \neq \mathit{nil} \Rightarrow R}$$

4.3 Operational semantics

The operational semantics is expressed in terms of the judgement

$$r, \sigma, \mu, S \vdash a \rightsquigarrow r', \sigma', \mu'.$$

It says that given an initial *operational state* (r, σ, μ) and *stack* S, executing command a terminates in operational state (r', σ', μ'). Operational states are triples whose first two components correspond to the register and data store components of states, as defined above. The third component is a *method store*. To define stacks and method stores, I first give some notational definitions.

For a partial function f from A to B, $a \in A$, and $b \in B$, I write $f.(a \mapsto b)$ to denote the function that is like f except possibly at a, which it maps to b. When $a_i \in A\ ^{i \in I}$ are distinct and $b_i \in B\ ^{i \in I}$, I write $(a_i \mapsto b_i\ ^{i \in I})$ for the partial function from A to B that maps a_i to b_i for $i \in I$ and is otherwise undefined. The partial function whose domain is empty is written $\emptyset$.

Let $\mathcal{H}$ denote a set of given *object names*. I assume $\mathcal{H} \cap \{\mathit{false}, \mathit{true}, \mathit{nil}\} = \emptyset$. A *stack* is a partial function from local variables to $\mathcal{H} \cup \{\mathit{false}, \mathit{true}, \mathit{nil}\}$. A *method store* is a partial function μ from $\mathcal{H}$, such that

- $\mu(h)(\mathsf{type})$ is the allocated type of object h, and
- $\mu(h)(\mathsf{m})$, if defined, is the implementation of method m of object h.

A *store pair* is a pair (σ, μ) where σ is a data store and μ is a method store.

In addition to keeping the method implementations of objects, the method store keeps the allocated type of objects. The operational semantics records this information as it allocates a new object, but doesn't use it subsequently. The information is used only to state and prove the soundness theorem. By conveniently recording this information in the operational semantics, where it causes no harm, one avoids the use of a *store type* [*cf.* Abadi and Leino 1997]. The result is a simpler statement and proof of soundness.

The rest of this subsection gives the rules of the operational semantics.

CONSTANT

$$\frac{}{r, \sigma, \mu, S \vdash c \rightsquigarrow c, \sigma, \mu}$$

LOCAL VARIABLE

$$\frac{r' = S(v)}{r, \sigma, \mu, S \vdash v \rightsquigarrow r', \sigma, \mu}$$

CONDITIONAL

$$\frac{\mathit{false}, \sigma, \mu, S \vdash a_0 \rightsquigarrow r', \sigma', \mu'}{\mathit{false}, \sigma, \mu, S \vdash a_0 \diamond a_1 \rightsquigarrow r', \sigma', \mu'} \qquad \frac{\mathit{true}, \sigma, \mu, S \vdash a_1 \rightsquigarrow r', \sigma', \mu'}{\mathit{true}, \sigma, \mu, S \vdash a_0 \diamond a_1 \rightsquigarrow r', \sigma', \mu'}$$

COMPOSITION

$$\frac{r, \sigma, \mu, S \vdash a_0 \rightsquigarrow r', \sigma', \mu' \qquad r', \sigma', \mu', S \vdash a_1 \rightsquigarrow r'', \sigma'', \mu''}{r, \sigma, \mu, S \vdash a_0\ ;\ a_1 \rightsquigarrow r'', \sigma'', \mu''}$$

Binding

$$\frac{r, \sigma, \mu, S.(v \mapsto r) \vdash a \rightsquigarrow r', \sigma', \mu'}{r, \sigma, \mu, S \vdash \mathit{with}\ v{:}T\ \mathit{do}\ a \rightsquigarrow r', \sigma', \mu'}$$

Object construction

$$\frac{\begin{array}{l} h \in \mathcal{H} \qquad \mathit{select}(\sigma, h, \mathsf{alloc}) = \mathit{false} \\ \sigma' = \mathit{store}(\cdots(\mathit{store}(\sigma, h, \mathsf{alloc}, \mathit{true}), \quad h, \mathsf{f}_i, c_i)\ ^{i \in I} \\ \mu' = \mu.(h \mapsto (\mathsf{type} \mapsto T,\ \mathsf{m}_j \mapsto a_j\ ^{j \in J})) \end{array}}{r, \sigma, \mu, S \vdash [T{:}\ \mathsf{f}_i = c_i\ ^{i \in I},\ \mathsf{m}_j = a_j\ ^{j \in J}] \rightsquigarrow h, \sigma', \mu'}$$

Field selection

$$\frac{r \in \mathcal{H} \qquad r' = \mathit{select}(\sigma, r, \mathsf{f})}{r, \sigma, \mu, S \vdash \mathsf{f} \rightsquigarrow r', \sigma, \mu}$$

Field update

$$\frac{r \in \mathcal{H} \qquad \sigma' = \mathit{store}(\sigma, r, \mathsf{f}, S(v))}{r, \sigma, \mu, S \vdash \mathsf{f} := v \rightsquigarrow r, \sigma', \mu}$$

Method invocation

$$\frac{r \in \mathcal{H} \qquad a = \mu(r)(\mathsf{m}) \qquad r, \sigma, \mu, \emptyset \vdash a \rightsquigarrow r', \sigma', \mu'}{r, \sigma, \mu, S \vdash \mathsf{m} \rightsquigarrow r', \sigma', \mu'}$$

5. Example

In this section, I show an example of a program that can be proved in the logic.

Let us consider a linked-list type with a method that appends a list to another. Reasoning about a program with such a type requires reasoning about reachability among linked-list nodes. To this end, we assume the underlying logic to contain a function symbol *Reach* (adapted from Greg Nelson's reachability predicate [Nelson 1983]). Informally,

$$\mathit{Reach}(e_0, e_1, \sigma, \mathsf{f}, e_2)$$

is *true* whenever it is possible to reach object e_1 from object e_0 via applications of f in σ, never going through object e_2.

The example in this section assumes that the underlying logic contains the following two axioms, which relate *Reach* to *select* and *store*, respectively.

$$\begin{array}{l} \langle \forall e_0, e_1, \sigma, \mathsf{f}, e_2 :: \\ \qquad \mathit{Reach}(e_0, e_1, \sigma, \mathsf{f}, e_2) = \mathit{true} \quad \equiv \\ \qquad e_0 = e_1 \vee (e_0 \neq e_2 \wedge \mathit{Reach}(\mathit{select}(\sigma, e_0, \mathsf{f}), e_1, \sigma, \mathsf{f}, e_2) = \mathit{true})\ \rangle \end{array} \tag{4}$$

$$\begin{array}{l} \langle \forall e_0, e_1, \sigma, \mathsf{f}_0, e_2, \mathsf{f}_1, e_3, e_4 :: \ \mathsf{f}_0 \neq \mathsf{f}_1 \Rightarrow \\ \qquad \mathit{Reach}(e_0, e_1, \sigma, \mathsf{f}_0, e_2) \ = \ \mathit{Reach}(e_0, e_1, \mathit{store}(\sigma, e_3, \mathsf{f}_1, e_4), \mathsf{f}_0, e_2)\ \rangle \end{array} \tag{5}$$

Axiom Eq. (4) resembles Nelson's axiom A1 and says that every object reaches itself, and that e_0 reaches e_1 if e_0 is not e_2 and $e_0.\mathsf{f}$ reaches e_1.

Axiom Eq. (5) says that whether or not e_0 reaches e_1 via f_0 is independent of the values of another field f_1.

The example uses the following environment, which I shall refer to as E:

$$\begin{array}{l} Node <: Object \\ \mathsf{next}: Node \to Node \\ \mathsf{appendArg}: Node \to Node \\ \mathsf{append}: Node \to Node : R \end{array}$$

where R is the relation

$$\begin{array}{lll} \grave{r} \neq nil & \Rightarrow & Reach(\grave{r}, \acute{r}, \grave{\sigma}, \mathsf{next}, nil) \wedge select(\grave{\sigma}, \acute{r}, \mathsf{next}) = nil \wedge \\ & & select(\acute{\sigma}, \acute{r}, \mathsf{next}) = select(\grave{\sigma}, \grave{r}, \mathsf{appendArg}) \wedge \\ & & \langle \forall\, o, \mathsf{f} :: \ select(\grave{\sigma}, o, \mathsf{f}) = select(\acute{\sigma}, o, \mathsf{f}) \vee \\ & & \qquad\qquad (o = \acute{r} \wedge \mathsf{f} = \mathsf{next}) \vee \mathsf{f} = \mathsf{appendArg} \,\rangle. \end{array}$$

Informally, this relation specifies that the method store its argument (which is passed in via the appendArg field) at the end of the list linked via field next. More precisely, the relation specifies that the method find an object $\acute{r}$ reachable from $\grave{r}$ (self) via the next field such that $\acute{r}.\mathsf{next}$ is nil. The method is to set $\acute{r}.\mathsf{next}$ to the given argument node. The method can modify $\acute{r}.\mathsf{next}$ and can modify the appendArg field of any object (since this field is used only as a way to pass a parameter to append anyway), but it is not allowed to modify the store in any other way.

To present the example code, I introduce a new command, *isnil*, which tests whether or not the register is *nil*.

NIL TEST

$$\frac{E, V \vdash \diamond \qquad E \vdash_{obj} T}{\begin{array}{l} E, V \vdash isnil : T \to Boolean : \\ \quad (\grave{r} = nil \;\Rightarrow\; \acute{r} = true) \wedge (\grave{r} \neq nil \;\Rightarrow\; \acute{r} = false) \wedge \grave{\sigma} = \acute{\sigma} \end{array}}$$

(Section 7 discusses expression commands such as *nil* tests.)

To write the program text, I assume the following binding powers, from highest to lowest:

$$:= \qquad . \qquad ; \qquad \lhd\!\rhd \qquad with\ \ldots\ do$$

Now, consider the following command.

$$\begin{array}{l} [Node\colon\ \mathsf{next} = nil,\ \mathsf{appendArg} = nil, \\ \qquad \mathsf{append} = with\ self\colon Node\ do\ \ \mathsf{appendArg}\ ;\ with\ n\colon Node\ do \\ \qquad\qquad self.\mathsf{next}\ ;\ isnil\ ; \\ \qquad\qquad (\ \ self.\mathsf{next}.\mathsf{appendArg} := n\ ;\ \mathsf{append} \\ \qquad\qquad \lhd\!\rhd\ self.\mathsf{next} := n\ ;\ self \\ \qquad\qquad) \\ ] \end{array} \tag{6}$$

This command allocates and returns a *Node* object whose fields, next and appendArg, are initially *nil*. The implementation of append starts by giving names to the self object (*self*) and the method's argument (n). Then it either calls append recursively on *self*.next or sets *self*.next to the given argument, depending on whether or not *self*.next is *nil*.

With axioms Eq. (4) and Eq. (5) in the underlying logic, one can prove the following judgement about the given allocation command.

$$E, \emptyset \vdash \text{Eq. (6)} : \mathit{Object} \to \mathit{Node} : \grave{r} = \acute{r} \tag{7}$$

Though the relation in this judgement ($\grave{r} = \acute{r}$) is trivial, establishing the judgement requires showing that the given implementation of append satisfies its declared specification. I omit the proof, which is straightforward.

I conclude the example with three remarks.

First, remember that to reason about a call to a method, in particular the recursive call to append, one uses the specification of the method being called, not its implementation. This makes the reasoning independent of the actual implementation of the callee, which may in fact even be a different implementation than the one shown (since each object can have its own implementation of its methods).

Second, remember that only partial correctness is proved. That is, Eq. (7) says that *if* the method terminates, its pre- and post-states will satisfy the specified relation. Indeed, an invocation of method append on an object in a cyclic structure of *Node* objects will not terminate.

Third, the static type of the fields *self*.next and *self*.appendArg is *Node*, but the dynamic types of these objects in an execution may be any subtype of *Node*. Note, however, that Eq. (7) is independent of the dynamic types of these objects. Indeed, having established Eq. (7) means that the method works in every execution. This is because the logic is *sound*, as is shown in the next section.

6. Soundness

This section states a soundness theorem, which proves the correctness of the operational semantics with respect to the axiomatic semantics. I first motivate the soundness theorem, and then state it together with an informal explanation. The appendix provides some additional formal definitions and gives the proof.

As a gentle step in presenting the full soundness theorem, consider the following theorem.

THEOREM 1. *If one can derive both* $E, \emptyset \vdash a : \mathit{Object} \to \mathit{Boolean} : \acute{r} = \mathit{true}$ *and* $\mathit{nil}, \sigma_0, \emptyset, \emptyset \vdash a \rightsquigarrow r, \sigma, \mu$*, then* $r = \mathit{true}$.

Here and in the next two theorems, σ_0 denotes a data store that satisfies

$$\langle \forall\, h \in \mathcal{H} \;::\; \mathit{select}(\sigma_0, h, \mathsf{alloc}) = \mathit{false}\; \rangle.$$

The theorem says that if in an environment E one can prove that a command a satisfies the transition relation $\acute{r} = true$, then any terminating execution of command a from a "reset" state ends with a register value of $true$.

A simple theorem about the result type of a command is the following.

THEOREM 2. *If one can derive* $E, \emptyset \vdash a : Object \rightarrow T : R$ *and* $E \vdash_{obj} T$ *and* $nil, \sigma_0, \emptyset, \emptyset \vdash a \rightsquigarrow r, \sigma, \mu$, *then the value* r *has type* T, *that is, either* $r = nil$ *or* $E \vdash \mu(r)(\mathsf{type}) <: T$.

This theorem says that if one can prove, using the axiomatic semantics, that a command a has final type T, where T is an object type, and one can show that, operationally, the program terminates with a register value of r, then r is a value of type T (that is, it is *nil* or its allocated type is a subtype of T). This theorem shows the soundness of the type system's treatment of object types.

An interesting theorem that says something about the final object store of a program is the following.

THEOREM 3. *If one can derive both* $E, \emptyset \vdash a : Object \rightarrow T : R$ *and* $nil, \sigma_0, \emptyset, \emptyset \vdash a \rightsquigarrow r, \sigma, \mu$, *then* $R[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := nil, \sigma_0, r, \sigma]$ *holds as a first-order predicate.*

This theorem says that if one can prove the two judgements about a, then relation R actually describes the relation between the initial and final states of each execution of a.

To prove the theorems above, one needs to prove something stronger. I call the stronger theorem, of which the theorems above are corollaries, the main soundness theorem:

THEOREM. (MAIN SOUNDNESS THEOREM) *If*

$$\begin{array}{lll} E, V \vdash a : T \rightarrow U : R & , \textit{ and} & (8) \\ r, \sigma, \mu, S \vdash a \rightsquigarrow r', \sigma', \mu' & , \textit{ and} & (9) \\ E, \sigma, \mu \Vdash r : T & , \textit{ and} & (10) \\ E \Vdash \sigma, \mu & , \textit{ and} & (11) \\ E, V, \sigma, \mu \Vdash S & , & (12) \end{array}$$

then

$$\begin{array}{lll} r, \sigma, r', \sigma', S \Vdash R & , \textit{ and} & (13) \\ (\sigma, \mu) \preceq (\sigma', \mu') & , \textit{ and} & (14) \\ E, \sigma', \mu' \Vdash r' : U & , \textit{ and} & (15) \\ E \Vdash \sigma', \mu' & . & (16) \end{array}$$

The appendix contains formal definitions of the judgements new to this theorem and a proof of the theorem. Here follows an informal explanation of the theorem and its components.

In the antecedent of the theorem, Eq. (8) and Eq. (9) express the judgements that have been derived for some command a. One can hope to say something interesting in the conclusion of the theorem only if the execution under consideration is from a "reasonable" state (r, σ, μ) and uses a "reasonable" stack S. Therefore, the judgement in Eq. (10) states that r is a value of type T, the judgement in Eq. (11) says that store pair (σ, μ) *matches* the environment E, and the judgement in Eq. (12) says that S is a *well-typed stack*.

In the conclusion of the theorem, Eq. (13) expresses that R does indeed describe the relation between the initial and final states of the execution, and Eq. (15) expresses that r' has type U. In addition, to use the theorem as a sufficiently strong induction hypothesis in the proof, Eq. (14) says that (σ, μ) is *continued by* (σ', μ'). This property expresses a kind of monotonicity that holds between two store pairs, the first of which precedes the other in some execution. Also, Eq. (16) says that (σ', μ'), like the initial store pair, matches the environment.

By removing Eq. (13) from the conclusion of the main theorem, one gets a corollary that expresses that the type system is sound with respect to the operational semantics. Such a corollary follows directly from the main theorem, but could also be proved directly in the same way that I prove the main theorem in the appendix.

7. Limitations of the logic

In this section, I discuss some limitations of the logic.

The object construction command is rather awkward. Because it lists method implementations, a method cannot directly construct objects whose type and method implementations are the same as for self. Instead, one can declare object types representing classes, as is done, for example, by Abadi and Cardelli [1996]. As an example, consider an object type *Seq* representing sequences of elements as a linked list. If *Seq* has a method extend that creates a new node and appends it to the end of the linked list, then one cannot write:

$$
\begin{array}{l}
[\mathit{Seq}\colon \ \ldots, \\
\qquad \mathsf{extend} = \mathit{with}\ \mathit{self}\colon \mathit{Seq}\ \mathit{do}\ \ldots\ [\mathit{Seq}\colon\ \ldots]\ \ldots \\
]
\end{array}
$$

because the inner object construction command must textually contain everything the outer one does. Rather, what works is to introduce an object type *SeqClass* with a method new that returns a new instance of type *Seq*, and to add a field class to *Seq*. One can then write:

$$
\begin{array}{l}
[\mathit{SeqClass}\colon\ \mathsf{new} = \mathit{with}\ \mathit{selfclass}\colon \mathit{SeqClass}\ \mathit{do} \\
\qquad\qquad [\mathit{Seq}\colon\ \mathsf{class} = \mathit{nil},\ \ldots, \\
\qquad\qquad\qquad \mathsf{extend} = \mathit{with}\ \mathit{self}\colon \mathit{Seq}\ \mathit{do}\ \ldots\ \mathit{self}.\mathsf{class}.\mathsf{new}\ \ldots \\
\qquad\qquad].\mathsf{class} := \mathit{selfclass} \\
].\mathsf{new}
\end{array}
$$

One can consider modifying the present logic to remove the limitation from the object construction command. For example, like in common class-based object-oriented languages, one can extend the program environment to include method implementations. One must then have a "link-time" check that ensures that every method that may be called by the program at run-time has an implementation. Or, like in common object-based languages, one can add a construct for cloning objects or their method implementations.

Another omission from the present logic is the ability to compare objects for equality. Just like one would expect to add primitive types like integers to the present logic, one would expect to add more general expressions, including comparison expressions. (It is actually possible, though quite cumbersome, to encode a mechanism by which the present logic can test whether or not a field is *nil*: Introduce a boolean field fIsNil for each object-valued field f, and then program so as to maintain the condition

$$(\mathsf{f} = nil \ \Rightarrow \ \mathsf{fIsNil} = true) \wedge (\mathsf{f} \neq nil \ \Rightarrow \ \mathsf{fIsNil} = false)$$

as an invariant of every object.)

A feature of the logic that to some readers may seem like a limitation is the fact that methods are given exactly one specification, not one specification for each type. That is, a subtype is not allowed to strengthen the specification of a method that it inherits from a supertype. This design choice simplifies matters, but does not hamper expressiveness. To illustrate, suppose that T is a type with a method m whose specification is R. Suppose further that U is a subtype of T, and that U's implementation of m, call it um, guarantees something stronger than R. Since methods are given only one specification, invocations of m on an object u are reasoned about (using the axiomatic semantics) as having the effect of R, even if u is statically known to be an object of type U. If the programmer who authored U is willing to let his clients rely on the stronger properties of the particular implementation um, he can do the following: declare a new method n, give n a sufficiently strong specification, implement n as um, and implement m as a call to n. Clients that must have the stronger specification now invoke n, not m. Much attention has been devoted to the related problem of co-variant result types, which can also be solved using this simple strategy.

A logic of programs provides a connection between programs and their specifications. In the present logic, method declarations contain specifications that are given simply as transition relations. Transition relations are not practically suited for writing down method specifications, because they are painfully explicit. Specification features like modifies clauses and abstract fields would remedy the situation, but lie outside the scope of this paper. To mention some work in this area, Lano and Haughton [1994] have surveyed object-oriented specifications, and my thesis [Leino 1995] shows how to deal with modifies clauses and data abstraction in modular, object-oriented programs. The logic for POOL [America and de Boer 1994] includes some specification features that can be used to state properties of recursive data structures.

8. Conclusion

I have presented a sound logic for object-oriented programs whose commands are imperative and whose objects are references to data fields. The programming notation requires that types, fields, and methods be declared in the environment before they can be used in a program. The main contributions of the paper are the logic itself, the soundness theorem, and the way that types are handled, which makes the subtype relation and the admission of recursive object types trivial.

Acknowledgements

I am grateful to Martín Abadi, Luca Cardelli, Greg Nelson, Raymie Stata, and the anonymous referees for helpful comments on the logic and the presentation thereof.

Appendix A. Proof of the main soundness theorem

Appendix A.1 Definitions

This subsection gives the definitions of the judgements used in the soundness theorem.

DEFINITION OF HOLDS. Judgement $r, \sigma, r', \sigma', S \Vdash R$ says that R is a relation that, when interpreted using r, σ, r', σ', and S, is a provable first-order formula. The judgement is defined as $\vdash_{fol} R[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma'][S]$, where $[S]$ denotes the following substitution: for every $v \in dom(S)$, every free occurrence of variable v is replaced by the result of the function application $S(v)$. I assume throughout that the domain of S does not contain values from $\mathcal{H} \cup \{false, true, nil\}$ (which I will denote h, c, or r, possibly primed), field names (like f or alloc), or data stores (which I will denote σ, possibly primed).

For example, to derive $r, \sigma, r', \sigma', S \Vdash select(\acute{\sigma}, \grave{r}, \mathsf{f}) = v$, where $v \in dom(S)$, one must establish $\vdash_{fol} select(\sigma', r, \mathsf{f}) = x$, where $x = S(v)$ (I will often write simply $\vdash_{fol} select(\sigma', r, \mathsf{f}) = S(v)$). Here, I assumed that none of σ', r, or f is in the domain of S.

DEFINITION OF FUNCTION EXTENSION. A function f' is called an *extension* of a function f, written $f \leq f'$, if $dom(f) \subseteq dom(f')$ and, for every $a \in dom(f)$, $f(a) = f'(a)$.

DEFINITION OF $\preceq$. A store pair (σ, μ) is said to be *continued by* a store pair (σ', μ'), written $(\sigma, \mu) \preceq (\sigma', \mu')$, when

C0 for every $h \in \mathcal{H}$, $select(\sigma, h, \mathsf{alloc}) = true \Rightarrow select(\sigma', h, \mathsf{alloc}) = true$, and

C1 $\mu \leq \mu'$.

Condition C0 says that any object allocated in σ remains allocated in σ', and C1 says that μ' contains the same type information and method implementations as μ does for those objects in $dom(\mu)$, but μ' may also contain information about objects not in $dom(\mu)$.

Note that $\preceq$ is reflexive and transitive.

DEFINITION OF HAS-TYPE. For any environment E, store pair (σ, μ), value r, and type T, the judgement

$$E, \sigma, \mu \Vdash r : T$$

says that T is a type in E and that r is allocated and has type T in (σ, μ). The judgement is derived using the following rules:

H0

$$\frac{E \vdash_{const} c{:}\, T}{E, \sigma, \mu \Vdash c : T}$$

H1

$$\frac{E \vdash_{obj} T \qquad select(\sigma, r, \mathsf{alloc}) = true \qquad \mu(r)(\mathsf{type}) = T}{E, \sigma, \mu \Vdash r : T}$$

H2

$$\frac{E, \sigma, \mu \Vdash r : U \qquad E \vdash U <: T}{E, \sigma, \mu \Vdash r : T}$$

Note that the judgement $E, \sigma, \mu \Vdash r : T$ is monotonic in (σ, μ) with respect to the $\preceq$ ordering: from $E, \sigma, \mu \Vdash r : T$ and $(\sigma, \mu) \preceq (\sigma', \mu')$, it follows that $E, \sigma', \mu' \Vdash r : T$. The judgement is also monotonic in T with respect to the $<:$ ordering: from $E, \sigma, \mu \Vdash r : T$ and $E \vdash T <: U$, it follows that $E, \sigma, \mu \Vdash r : U$.

Observe that any derivation of $E, \sigma, \mu \Vdash r : T$ consists of exactly one application of H0 or H1, followed by some number of applications of H2. For any object type T, rule H1 (not H0) is applied when $r \in \mathcal{H}$, since $\mathcal{H}$ does not contain *nil*. We arrive at the following lemma, which will be useful in the proof:

LEMMA. (ABOUT HAS-TYPE) *If* $E \vdash_{obj} T$, $E, \sigma, \mu \Vdash r : T$, *and* $r \in \mathcal{H}$, *then* $select(\sigma, r, \mathsf{alloc}) = true$ *and* $E \vdash \mu(r)(\mathsf{type}) <: T$.

DEFINITION OF MATCHING. For any environment E and store pair (σ, μ), the judgement $E \Vdash \sigma, \mu$ says that (σ, μ) *matches* environment E. The matching judgement holds when all of the following hold:

M0 for every $h \in \mathcal{H}$, $select(\sigma, h, \mathsf{alloc}) = true \quad \equiv \quad h \in dom(\mu)$

M1 for every $h \in dom(\mu)$, $\mu(h)(\mathsf{type})$ is defined

M2 for all h, f, T, U, such that $h \in \mathcal{H}$, $select(\sigma, h, \mathsf{alloc}) = true$, and $\mathsf{f}{:}T \to U \in Fields(\mu(h)(\mathsf{type}), E)$,

$$E, \sigma, \mu \Vdash select(\sigma, h, \mathsf{f}) : U$$

M3 for all h, m, T, U, R, such that $h \in \mathcal{H}$, $select(\sigma, h, \mathsf{alloc}) = true$, and $\mathsf{m}{:}T \to U : R \in Methods(\mu(h)(\mathsf{type}), E)$,

$$E, \emptyset \vdash \mu(h)(\mathsf{m}) : T \to U : R.$$

Rule M0 states that the objects allocated in σ are the objects about which μ contains information, and M1 states that μ contains type information for each of those objects. Rule M2 says that every declared field of an allocated object has the appropriate type. Rule M3 says that every declared method of an allocated object has an implementation in μ that satisfies the declared method specification.

Definition of well-typed stack. The definition of a well-typed stack uses a function *ids*. For V a list of local variable declarations (like $v{:}T$), $ids(V)$ denotes the set of identifiers (that is, v) of those declarations.

For any environment (E, V), store pair (σ, μ), and stack S, the judgement $E, V, \sigma, \mu \Vdash S$ says that S is *well-typed* with respect to V. The judgement holds when

S0 $dom(S) = ids(V)$, and

S1 $E, \sigma, \mu \Vdash S(v) : T$ holds for every $v{:}T$ pair in V.

Note that, because has-type is monotonic in (σ, μ), so is the well-typed stack judgement.

Appendix A.2 Proof

This subsection gives the proof of the main soundness theorem. The proof is by induction on the lexicographic pair (d, e), where d is the derivation of the operational-semantics judgement about command a, and e is the axiomatic-semantics derivation of the judgement about a. The proof is a case study of the last rule applied in derivation e, a strategy that avoids the need to consider possible applications of the axiomatic subsumption rule in the case of each program construct.

Subsumption. We're given

$$\begin{array}{lll} E, V \vdash a : T_0 \to T_3 : R' & r, \sigma, \mu, S \vdash a \rightsquigarrow r', \sigma', \mu' & \\ E, \sigma, \mu \Vdash r : T_0 & E \Vdash \sigma, \mu & E, V, \sigma, \mu \Vdash S. \end{array}$$

Since this case considers subsumption, we are given that the first of these formulas has been derived from

$$\begin{array}{llll} E, V \vdash a : T_1 \to T_2 : R & & & \\ E \vdash T_0 <: T_1 & E \vdash T_2 <: T_3 & \vdash_{fol} R \Rightarrow R' & E, V \vdash_{rel} R'. \end{array}$$

We attempt to invoke the induction hypothesis with $T, U, R := T_1, T_2, R$. To do so, we must first establish

$$E, \sigma, \mu \Vdash r : T_1$$

which follows from $E, \sigma, \mu \Vdash r : T_0$, $E \vdash T_0 <: T_1$, and the monotonicity of has-type. By the induction hypothesis, we then have

$$r, \sigma, r', \sigma', S \Vdash R \qquad (\sigma, \mu) \preceq (\sigma', \mu') \qquad E, \sigma', \mu' \Vdash r' : T_2 \qquad E \Vdash \sigma', \mu'.$$

Now for our proof obligations:

- $r, \sigma, r', \sigma', S \Vdash R'$. Follows from $r, \sigma, r', \sigma', S \Vdash R$, $\vdash_{fol} R \Rightarrow R'$, and the monotonicity of holds.
- $(\sigma, \mu) \preceq (\sigma', \mu')$. Follows directly from induction hypothesis.
- $E, \sigma', \mu' \Vdash r' : T_3$. Follows from $E, \sigma', \mu' \Vdash r' : T_2$, $E \vdash T_2 <: T_3$, and the monotonicity of has-type.
- $E \Vdash \sigma', \mu'$. Follows directly from induction hypothesis.

Constant. In this case and all remaining cases, the axiomatic semantics rule considered in the case determines the shape of the command, which in turn determines which kind of operational semantics rule has been applied.

We're given

$$\begin{array}{l} E, V \vdash c : U \to T : \acute{r} = c \wedge \grave{\sigma} = \acute{\sigma} \qquad r, \sigma, \mu, S \vdash c \rightsquigarrow c, \sigma, \mu \\ E, \sigma, \mu \Vdash r : U \qquad E \Vdash \sigma, \mu \qquad E, V, \sigma, \mu \Vdash S \end{array}$$

derived from

$$E, V \vdash \diamond \qquad E \vdash_{const} c{:}\, T \qquad E \vdash_{type} U.$$

We prove:

- $$\begin{array}{ll} & r, \sigma, c, \sigma, S \Vdash \acute{r} = c \wedge \grave{\sigma} = \acute{\sigma} \\ = & \quad \{ \text{ definition of holds } \} \\ & \vdash_{fol} (\acute{r} = c \wedge \grave{\sigma} = \acute{\sigma})[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, c, \sigma][S] \\ = & \quad \{ \text{ substitution } \} \\ & \vdash_{fol} c = c \wedge \sigma = \sigma \end{array}$$

  which is a tautology.
- $(\sigma, \mu) \preceq (\sigma, \mu)$. $\preceq$ is reflexive.
- $E, \sigma, \mu \Vdash c : T$. Follows from $E \vdash_{const} c{:}\, T$ and H0 of the definition of has-type.
- $E \Vdash \sigma, \mu$. Given.

LOCAL VARIABLE. We're given

$$E, V \vdash v : U \rightarrow T : \acute{r} = v \wedge \grave{\sigma} = \acute{\sigma} \qquad r, \sigma, \mu, S \vdash v \rightsquigarrow r', \sigma, \mu$$
$$E, \sigma, \mu \Vdash r : U \qquad E \Vdash \sigma, \mu \qquad E, V, \sigma, \mu \Vdash S$$

where the axiomatic-semantics judgement has been derived from

$$E, V \vdash_{var} v{:}T \qquad E \vdash_{type} U$$

and the operational-semantics judgement has been derived from

$$r' = S(v).$$

We prove:

- $\quad r, \sigma, r', \sigma, S \Vdash \acute{r} = v \wedge \grave{\sigma} = \acute{\sigma}$
  $=\qquad \{$ definition of holds $\}$
  $\quad \vdash_{fol} (\acute{r} = v \wedge \grave{\sigma} = \acute{\sigma})[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma][S]$
  $=\qquad \{$ substitution $\}$
  $\quad \vdash_{fol} r' = S(v) \wedge \sigma = \sigma$
  which follows from $r' = S(v)$.
- $(\sigma, \mu) \preceq (\sigma, \mu)$. $\preceq$ is reflexive.
- $E, \sigma, \mu \Vdash r' : T$. Follows from $r' = S(v)$, $E, V \vdash_{var} v{:}T$, and the definition of $E, V, \sigma, \mu \Vdash S$.
- $E \Vdash \sigma, \mu$. Given.

CONDITIONAL. There are two subcases, because the operational-semantics judgement can be derived from one of two rules. I show the proof for the *false*-rule; the *true*-rule is similar. We're given

$$E, V \vdash a_0 \diamond a_1 : Boolean \rightarrow T : (\grave{r} = false \Rightarrow R_0) \wedge (\grave{r} = true \Rightarrow R_1)$$
$$false, \sigma, \mu, S \vdash a_0 \diamond a_1 \rightsquigarrow r', \sigma', \mu'$$
$$E, \sigma, \mu \Vdash false : Boolean \qquad E \Vdash \sigma, \mu \qquad E, V, \sigma, \mu \Vdash S$$

derived from

$$E, V \vdash a_0 : Boolean \rightarrow T : R_0 \qquad E, V \vdash a_1 : Boolean \rightarrow T : R_1$$

and

$$false, \sigma, \mu, S \vdash a_0 \rightsquigarrow r', \sigma', \mu'.$$

By the induction hypothesis applied on a_0, we have:

$$false, \sigma, r', \sigma', S \Vdash R_0 \qquad (\sigma, \mu) \preceq (\sigma', \mu')$$
$$E, \sigma', \mu' \Vdash r' : T \qquad E \Vdash \sigma', \mu'.$$

We prove:

$$
\begin{array}{ll}
\circ & \mathit{false}, \sigma, r', \sigma', S \Vdash (\grave{r} = \mathit{false} \;\Rightarrow\; R_0) \wedge (\grave{r} = \mathit{true} \;\Rightarrow\; R_1) \\
= & \qquad \{ \quad \text{definition of holds} \quad \} \\
 & \vdash_{\mathit{fol}} ((\grave{r} = \mathit{false} \;\Rightarrow\; R_0) \wedge (\grave{r} = \mathit{true} \;\Rightarrow\; R_1)) \\
 & \qquad\qquad [\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := \mathit{false}, \sigma, r', \sigma'][S] \\
= & \qquad \{ \quad \text{substitution} \quad \} \\
 & \vdash_{\mathit{fol}} (\mathit{false} = \mathit{false} \;\Rightarrow\; R_0[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := \mathit{false}, \sigma, r', \sigma'][S]) \;\wedge \\
 & \qquad (\mathit{false} = \mathit{true} \;\Rightarrow\; R_1[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := \mathit{false}, \sigma, r', \sigma'][S]) \\
= & \qquad \{ \quad \text{propositional calculus, using axiom Eq. (2):} \\
 & \qquad\qquad \mathit{false} \neq \mathit{true} \quad \} \\
 & \vdash_{\mathit{fol}} R_0[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := \mathit{false}, \sigma, r', \sigma'][S] \\
= & \qquad \{ \quad \text{definition of holds} \quad \} \\
 & \mathit{false}, \sigma, r', \sigma', S \Vdash R_0
\end{array}
$$

which comes directly from the induction hypothesis.

- $(\sigma, \mu) \preceq (\sigma', \mu')$. Directly from induction hypothesis.
- $E, \sigma', \mu' \Vdash r' : T$. Directly from induction hypothesis.
- $E \Vdash \sigma', \mu'$. Directly from induction hypothesis.

Composition. We're given

$$
\begin{array}{l}
E, V \vdash a_0 \;;\; a_1 : T_0 \to T_2 : \langle \exists \check{r}, \check{\sigma} :: R_0[\acute{r}, \acute{\sigma} := \check{r}, \check{\sigma}] \wedge R_1[\grave{r}, \grave{\sigma} := \check{r}, \check{\sigma}] \rangle \\
r, \sigma, \mu, S \vdash a_0 \;;\; a_1 \rightsquigarrow r'', \sigma'', \mu'' \\
E, \sigma, \mu \Vdash r : T_0 \qquad E \Vdash \sigma, \mu \qquad E, V, \sigma, \mu \Vdash S
\end{array}
$$

derived from

$$
\begin{array}{l}
E, V \vdash a_0 : T_0 \to T_1 : R_0 \qquad E, V \vdash a_1 : T_1 \to T_2 : R_1 \\
\check{r} \text{ and } \check{\sigma} \text{ do not occur free in } R_0 \text{ or } R_1
\end{array}
$$

and

$$
r, \sigma, \mu, S \vdash a_0 \rightsquigarrow r', \sigma', \mu' \qquad r', \sigma', \mu', S \vdash a_1 \rightsquigarrow r'', \sigma'', \mu''.
$$

Applying the induction hypothesis on a_0, we get:

$$
\begin{array}{ll}
r, \sigma, r', \sigma', S \Vdash R_0 & (\sigma, \mu) \preceq (\sigma', \mu') \\
E, \sigma', \mu' \Vdash r' : T_1 & E \Vdash \sigma', \mu'.
\end{array}
$$

We wish to invoke the induction hypothesis also on a_1. This requires that we first establish

$$
E, \sigma', \mu' \Vdash r' : T_1 \qquad E \Vdash \sigma', \mu' \qquad E, V, \sigma', \mu' \Vdash S.
$$

The first two of these judgements come directly from the application of the induction hypothesis on a_0; the third follows from $E, V, \sigma, \mu \Vdash S$, $(\sigma, \mu) \preceq (\sigma', \mu')$, and the monotonicity of well-typed stacks. By the induction hypothesis applied to a_1, we now have:

$$
\begin{array}{ll}
r', \sigma', r'', \sigma'', S \Vdash R_1 & (\sigma', \mu') \preceq (\sigma'', \mu'') \\
E, \sigma'', \mu'' \Vdash r'' : T_2 & E \Vdash \sigma'', \mu''.
\end{array}
$$

Now for the proof obligations:

$$
\begin{array}{ll}
\circ & r,\sigma,r'',\sigma'',S \Vdash \langle\, \exists\, \check{r},\check{\sigma} \;::\; R_0[\acute{r},\acute{\sigma} := \check{r},\check{\sigma}] \wedge R_1[\grave{r},\grave{\sigma} := \check{r},\check{\sigma}] \,\rangle \\
= & \qquad \{ \text{ definition of holds } \} \\
 & \vdash_{fol} \langle\, \exists\, \check{r},\check{\sigma} \;::\; R_0[\acute{r},\acute{\sigma} := \check{r},\check{\sigma}] \wedge R_1[\grave{r},\grave{\sigma} := \check{r},\check{\sigma}] \,\rangle \\
 & \qquad\qquad [\grave{r},\grave{\sigma},\acute{r},\acute{\sigma} := r,\sigma,r'',\sigma''][S] \\
\Leftarrow & \qquad \{ \text{ instantiate } \check{r},\check{\sigma} := r',\sigma', \text{ since substitution is} \\
 & \qquad\quad \text{monotonic } \} \\
 & \vdash_{fol} (R_0[\acute{r},\acute{\sigma} := \check{r},\check{\sigma}] \wedge R_1[\grave{r},\grave{\sigma} := \check{r},\check{\sigma}])[\check{r},\check{\sigma} := r',\sigma'] \\
 & \qquad\qquad [\grave{r},\grave{\sigma},\acute{r},\acute{\sigma} := r,\sigma,r'',\sigma''][S] \\
= & \qquad \{ \text{ substitution, since } R_0 \text{ and } R_1 \text{ are free of } \check{r} \text{ and } \check{\sigma} \ \} \\
 & \vdash_{fol} R_0[\grave{r},\grave{\sigma},\acute{r},\acute{\sigma} := r,\sigma,r',\sigma'][S] \wedge \\
 & \qquad R_1[\grave{r},\grave{\sigma},\acute{r},\acute{\sigma} := r',\sigma',r'',\sigma''][S] \\
= & \qquad \{ \text{ satisfiability distributes over } \wedge \ \} \\
 & \vdash_{fol} R_0[\grave{r},\grave{\sigma},\acute{r},\acute{\sigma} := r,\sigma,r',\sigma'][S] \\
 & \vdash_{fol} R_1[\grave{r},\grave{\sigma},\acute{r},\acute{\sigma} := r',\sigma',r'',\sigma''][S] \\
= & \qquad \{ \text{ definition of holds, twice } \} \\
 & r,\sigma,r',\sigma',S \Vdash R_0 \\
 & r',\sigma',r'',\sigma'',S \Vdash R_1
\end{array}
$$

which come directly from the induction hypotheses.

- $(\sigma',\mu') \preceq (\sigma'',\mu'')$. Follows from $(\sigma,\mu) \preceq (\sigma',\mu')$ and $(\sigma',\mu') \preceq (\sigma'',\mu'')$, and transitivity of $\preceq$.
- $E,\sigma'',\mu'' \Vdash r'' : T_2$. Follows directly from the induction hypothesis applied to a_1.
- $E \Vdash \sigma'',\mu''$. Follows directly from the induction hypothesis applied to a_1.

Binding. We're given

$$
\begin{array}{l}
E,V \vdash \textit{with } v{:}T \textit{ do } a : T \to U : R[v := \grave{r}] \\
r,\sigma,\mu,S \vdash \textit{with } v{:}T \textit{ do } a \rightsquigarrow r',\sigma',\mu' \\
E,\sigma,\mu \Vdash r : T \qquad E \Vdash \sigma,\mu \qquad E,V,\sigma,\mu \Vdash S
\end{array}
$$

derived from

$$E,(V,v{:}T) \vdash a : T \to U : R$$

and

$$r,\sigma,\mu,S.(v \mapsto r) \vdash a \rightsquigarrow r',\sigma',\mu'.$$

We'd like to apply the induction hypothesis on command a, transition relation R, command environment $(E,(V,v{:}T))$, and stack $S.(v \mapsto r)$. In order to do so, we must first establish

$$E,(V,v{:}T),\sigma,\mu \Vdash S.(v \mapsto r)$$

which follows from $E,V,\sigma,\mu \Vdash S$, $E,\sigma,\mu \Vdash (S.(v \mapsto r))(v) : T$, and the definition of well-typed stacks, since $(S.(v \mapsto r))(v) = r$ and we're given $E,\sigma,\mu \Vdash r : T$. Invoking the induction hypothesis as planned, we obtain:

$$
\begin{array}{l}
r,\sigma,r',\sigma',S.(v \mapsto r) \Vdash R \qquad (\sigma,\mu) \preceq (\sigma',\mu') \\
E,\sigma',\mu' \Vdash r' : U \qquad E \Vdash \sigma',\mu'.
\end{array}
$$

Now:

$$
\begin{array}{ll}
\circ & r, \sigma, r', \sigma', S \Vdash R[v := \grave{r}] \\
= & \quad \{ \text{ definition of holds } \} \\
& \vdash_{\mathit{fol}} R[v := \grave{r}][\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma'][S] \\
= & \quad \{ \text{ } v \text{ does not occur free in } R[v := \grave{r}][\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma'], \\
& \quad \text{ the argument to } [S] \ \} \\
& \vdash_{\mathit{fol}} R[v := \grave{r}][\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma'][S.(v \mapsto r)] \\
\Leftarrow & \quad \{ \text{ substitution is weakening and monotonic } \} \\
& \vdash_{\mathit{fol}} R[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma'][S.(v \mapsto r)] \\
= & \quad \{ \text{ definition of holds } \} \\
& r, \sigma, r', \sigma', S.(v \mapsto r) \Vdash R
\end{array}
$$

which comes directly from the induction hypothesis.

- $(\sigma, \mu) \preceq (\sigma', \mu')$. Directly from induction hypothesis.
- $E, \sigma', \mu' \Vdash r' : U$. Directly from induction hypothesis.
- $E \Vdash \sigma', \mu'$. Directly from induction hypothesis.

Object construction. We're given

$$
\begin{array}{l}
E, V \vdash [T\!: \ \mathsf{f}_i = c_i \ {}^{i \in I}, \ \mathsf{m}_j = a_j \ {}^{j \in J}] : U \to T : \\
\qquad \acute{r} \neq \mathit{nil} \wedge \mathit{select}(\grave{\sigma}, \acute{r}, \mathsf{alloc}) = \mathit{false} \ \wedge \\
\qquad \acute{\sigma} = \mathit{store}(\cdots(\mathit{store}(\grave{\sigma}, \acute{r}, \mathsf{alloc}, \mathit{true}), \quad \acute{r}, \mathsf{f}_i, c_i) \ {}^{i \in I} \\
r, \sigma, \mu, S \vdash [T\!: \ \mathsf{f}_i = c_i \ {}^{i \in I}, \ \mathsf{m}_j = a_j \ {}^{j \in J}] \rightsquigarrow h, \sigma', \mu' \\
E, \sigma, \mu \Vdash r : U \qquad E \Vdash \sigma, \mu \qquad E, V, \sigma, \mu \Vdash S
\end{array}
$$

derived from

$$
\begin{array}{l}
E, V \vdash \diamond \qquad E \vdash_{\mathit{type}} U \qquad E \vdash_{\mathit{obj}} T \\
\mathsf{f}_i\!: T_i \to U_i \ {}^{i \in I} \text{ are the elements of } \mathit{Fields}(T, E) \\
\mathsf{m}_j\!: T_j \to U_j : R_j \ {}^{j \in J} \text{ are the elements of } \mathit{Methods}(T, E) \\
E \vdash_{\mathit{const}} c_i\!: U_i \ {}^{i \in I} \qquad E, \emptyset \vdash a_j : T \to U_j : R_j \ {}^{j \in J}
\end{array}
$$

and

$$
\begin{array}{l}
h \in \mathcal{H} \qquad \mathit{select}(\sigma, h, \mathsf{alloc}) = \mathit{false} \\
\sigma' = \mathit{store}(\cdots(\mathit{store}(\sigma, h, \mathsf{alloc}, \mathit{true}), \quad h, \mathsf{f}_i, c_i) \ {}^{i \in I} \\
\mu' = \mu.(h \mapsto (\mathsf{type} \mapsto T, \ \mathsf{m}_j \mapsto a_j \ {}^{j \in J})).
\end{array}
$$

We prove:

$$
\begin{array}{ll}
\circ & r, \sigma, h, \sigma', S \Vdash \acute{r} \neq \mathit{nil} \wedge \mathit{select}(\grave{\sigma}, \acute{r}, \mathsf{alloc}) = \mathit{false} \ \wedge \\
& \qquad \acute{\sigma} = \mathit{store}(\cdots(\mathit{store}(\grave{\sigma}, \acute{r}, \mathsf{alloc}, \mathit{true}), \quad \acute{r}, \mathsf{f}_i, c_i) \ {}^{i \in I} \\
= & \quad \{ \text{ definition of holds } \} \\
& \vdash_{\mathit{fol}} (\acute{r} \neq \mathit{nil} \wedge \mathit{select}(\grave{\sigma}, \acute{r}, \mathsf{alloc}) = \mathit{false} \ \wedge \\
& \qquad \acute{\sigma} = \mathit{store}(\cdots(\mathit{store}(\grave{\sigma}, \acute{r}, \mathsf{alloc}, \mathit{true}), \quad \acute{r}, \mathsf{f}_i, c_i) \ {}^{i \in I}) \\
& \qquad\quad [\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, h, \sigma'][S] \\
= & \quad \{ \text{ substitution } \} \\
& \vdash_{\mathit{fol}} h \neq \mathit{nil} \wedge \mathit{select}(\sigma, h, \mathsf{alloc}) = \mathit{false} \ \wedge \\
& \qquad \sigma' = \mathit{store}(\cdots(\mathit{store}(\sigma, h, \mathsf{alloc}, \mathit{true}), \quad h, \mathsf{f}_i, c_i) \ {}^{i \in I}
\end{array}
$$

which follows directly from $\mathit{nil} \notin \mathcal{H}$ and the way the hypothesis is derived.

- $(\sigma, \mu) \preceq (\sigma', \mu')$. Re C0: For any $k \in \mathcal{H}$, $select(\sigma, k, \mathsf{alloc})$ differs from $select(\sigma', k, \mathsf{alloc})$ only for $k = h$, and we're given the equality $select(\sigma, h, \mathsf{alloc}) = false$. Re C1: To prove $\mu \leq \mu'$, it suffices to show $h \notin dom(\mu)$, which follows from $select(\sigma, h, \mathsf{alloc}) = false$ and M0 in the definition of the given judgement $E \Vdash \sigma, \mu$.

- $E, \sigma', \mu' \Vdash h : T$. Follows from the given $E \vdash_{obj} T$, the facts $select(\sigma', h, \mathsf{alloc}) = true$ and $\mu'(h)(\mathsf{type}) = T$, and rule H1 in the definition of has-type.

- $E \Vdash \sigma', \mu'$. Follows from $E \Vdash \sigma, \mu$ and the following observations. Re M0: We have $select(\sigma', h, \mathsf{alloc}) = true$ and $h \in dom(\mu')$. Re M1: $\mu'(h)(\mathsf{type})$ is defined. Re M2: For each appropriate f_i, we have $select(\sigma', h, \mathsf{f}_i) = c_i$, so we need to show $E, \sigma', \mu' \Vdash c_i : U_i$, which follows from H0 and $E \vdash_{const} c_i : U_i$. Re M3: For each appropriate m_j, we have $\mu'(h)(\mathsf{m}_j) = a_j$, and we're given $E, \emptyset \vdash a_j : T \to U_j : R_j$.

Field selection. We're given

$$
\begin{array}{l}
E, V \vdash \mathsf{f} : T \to U : \grave{r} \neq nil \Rightarrow \acute{r} = select(\grave{\sigma}, \grave{r}, \mathsf{f}) \wedge \grave{\sigma} = \acute{\sigma} \\
r, \sigma, \mu, S \vdash \mathsf{f} \rightsquigarrow r', \sigma, \mu \\
E, \sigma, \mu \Vdash r : T \qquad E \Vdash \sigma, \mu \qquad E, V, \sigma, \mu \Vdash S
\end{array}
$$

derived from

$$E, V \vdash \diamond \qquad E \vdash_{field} \mathsf{f} : T \to U$$

and

$$r \in \mathcal{H} \qquad r' = select(\sigma, r, \mathsf{f}).$$

Now:

- $$
\begin{array}{ll}
 & r, \sigma, r', \sigma, S \Vdash \grave{r} \neq nil \Rightarrow \acute{r} = select(\grave{\sigma}, \grave{r}, \mathsf{f}) \wedge \grave{\sigma} = \acute{\sigma} \\
= & \quad \{ \text{ definition of holds } \} \\
 & \vdash_{fol} (\grave{r} \neq nil \Rightarrow \acute{r} = select(\grave{\sigma}, \grave{r}, \mathsf{f}) \wedge \grave{\sigma} = \acute{\sigma}) \\
 & \qquad [\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma][S] \\
= & \quad \{ \text{ substitution } \} \\
 & \vdash_{fol} r \neq nil \Rightarrow r' = select(\sigma, r, \mathsf{f}) \wedge \sigma = \sigma
\end{array}
$$

  which holds because $r \in \mathcal{H}$, $nil \notin \mathcal{H}$, and $r' = select(\sigma, r, \mathsf{f})$.

- $(\sigma, \mu) \preceq (\sigma, \mu)$. $\preceq$ is reflexive.

- $E, \sigma, \mu \Vdash r' : U$. From the way $E \vdash_{field} \mathsf{f} : T \to U$ was derived, we have $E \vdash_{obj} T$. Thus, from $E, \sigma, \mu \Vdash r : T$, $r \in \mathcal{H}$, and the lemma about has-type, we have $select(\sigma, r, \mathsf{alloc}) = true$ and $E \vdash \mu(r)(\mathsf{type}) <: T$. Thus by M2 in the definition of the given $E \Vdash \sigma, \mu$, we have $E, \sigma, \mu \Vdash select(\sigma, r, \mathsf{f}) : U$. Since $r' = select(\sigma, r, \mathsf{f})$, the proof obligation follows.

- $E \Vdash \sigma, \mu$. Given.

FIELD UPDATE. We're given

$$E, V \vdash \mathsf{f} := v : T_1 \to T_1 : \grave{r} \neq \mathit{nil} \Rightarrow \grave{r} = \acute{r} \wedge \acute{\sigma} = \mathit{store}(\grave{\sigma}, \grave{r}, \mathsf{f}, v)$$
$$r, \sigma, \mu, S \vdash \mathsf{f} := v \rightsquigarrow r, \sigma', \mu$$
$$E, \sigma, \mu \Vdash r : T_1 \qquad E \Vdash \sigma, \mu \qquad E, V, \sigma, \mu \Vdash S$$

derived from

$$\begin{array}{ll} E \vdash_{\mathit{field}} \mathsf{f}{:}\, T_0 \to U_0 & E \vdash T_1 <: T_0 \\ E, V \vdash_{\mathit{var}} v{:}\, U_1 & E \vdash U_1 <: U_0 \end{array}$$

and

$$r \in \mathcal{H} \qquad \sigma' = \mathit{store}(\sigma, r, \mathsf{f}, S(v)).$$

Now:

- $$\begin{array}{ll} & r, \sigma, r, \sigma', S \Vdash \grave{r} \neq \mathit{nil} \Rightarrow \grave{r} = \acute{r} \wedge \acute{\sigma} = \mathit{store}(\grave{\sigma}, \grave{r}, \mathsf{f}, v) \\ = & \quad \{ \text{ definition of holds } \} \\ & \vdash_{\mathit{fol}} (\grave{r} \neq \mathit{nil} \Rightarrow \grave{r} = \acute{r} \wedge \acute{\sigma} = \mathit{store}(\grave{\sigma}, \grave{r}, \mathsf{f}, v)) \\ & \qquad [\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r, \sigma'][S] \\ = & \quad \{ \text{ substitution } \} \\ & \vdash_{\mathit{fol}} r \neq \mathit{nil} \Rightarrow r = r \wedge \sigma' = \mathit{store}(\sigma, r, \mathsf{f}, S(v)) \end{array}$$

  which follows from the way the hypothesis is derived.

- $(\sigma, \mu) \preceq (\sigma', \mu)$. Follows from the fact that alloc is unchanged: for all h, $\mathit{select}(\sigma, h, \mathsf{alloc}) = \mathit{select}(\sigma', h, \mathsf{alloc})$.

- $E, \sigma', \mu \Vdash r : T_1$. Follows from $E, \sigma, \mu \Vdash r : T_1$, $(\sigma, \mu) \preceq (\sigma', \mu)$, and the monotonicity of has-type.

- $E \Vdash \sigma', \mu$. Because of $E \Vdash \sigma, \mu$ and the observation above that no alloc field has changed between σ and σ', it suffices to show $E, \sigma', \mu \Vdash \mathit{select}(\sigma', f, \mathsf{f}) : U_0$, which we do by the following calculation that starts from two givens:

  $$\begin{array}{ll} & E, V, \sigma, \mu \Vdash S \qquad E, V \vdash_{\mathit{var}} v{:}\, U_1 \\ \Rightarrow & \quad \{ \text{ definition of well-typed stack } \} \\ & E, \sigma, \mu \Vdash S(v) : U_1 \\ \Rightarrow & \quad \{ \; E \vdash U_1 <: U_0 \text{ and H2 } \} \\ & E, \sigma, \mu \Vdash S(v) : U_0 \\ \Rightarrow & \quad \{ \; (\sigma, \mu) \preceq (\sigma', \mu) \text{ and monotonicity of has-type } \} \\ & E, \sigma', \mu \Vdash S(v) : U_0 \\ = & \quad \{ \; \mathit{select}/\mathit{store} \text{ axiom (Eq. (1)) } \} \\ & E, \sigma', \mu \Vdash \mathit{select}(\mathit{store}(\sigma, r, \mathsf{f}, S(v)), r, \mathsf{f}) : U_0 \\ = & \quad \{ \; \sigma' = \mathit{store}(\sigma, r, \mathsf{f}, S(v)) \; \} \\ & E, \sigma', \mu \Vdash \mathit{select}(\sigma', r, \mathsf{f}) : U_0. \end{array}$$

METHOD INVOCATION. We're given

$$E, V \vdash \mathsf{m} : T \to U : \grave{r} \neq nil \;\Rightarrow\; R \qquad r, \sigma, \mu, S \vdash \mathsf{m} \rightsquigarrow r', \sigma', \mu'$$
$$E, \sigma, \mu \Vdash r : T \qquad E \Vdash \sigma, \mu \qquad E, V, \sigma, \mu \Vdash S$$

derived from

$$E, V \vdash \diamond \qquad E \vdash_{method} \mathsf{m}\colon T \to U : R$$

and

$$r \in \mathcal{H} \qquad a = \mu(r)(\mathsf{m}) \qquad r, \sigma, \mu, \emptyset \vdash a \rightsquigarrow r', \sigma', \mu'.$$

We wish to apply the induction hypothesis on command a, relation R, and with $V, S := \emptyset, \emptyset$. This requires that we first establish $E, \emptyset, \sigma, \mu \Vdash \emptyset$, which holds trivially, and

$$\begin{array}{ll}
 & E, \emptyset \vdash a : T \to U : R \\
\Leftarrow & \qquad \{ \;\; a = \mu(r)(\mathsf{m}), \text{ and M3 of } E \Vdash \sigma, \mu \;\; \} \\
 & select(\sigma, r, \mathsf{alloc}) = true \qquad r \in \mathcal{H} \\
 & \mathsf{m}\colon T \to U : R \in Methods(\mu(r)(\mathsf{type}), E), \\
\Leftarrow & \qquad \{ \;\; E \vdash_{method} \mathsf{m}\colon T \to U : R \text{ and definition of } Methods \;\; \} \\
 & select(\sigma, r, \mathsf{alloc}) = true \qquad r \in \mathcal{H} \\
 & E \vdash \mu(r)(\mathsf{type}) <: T \\
\Leftarrow & \qquad \{ \;\; \text{lemma about has-type} \;\; \} \\
 & E, \sigma, \mu \Vdash r : T \qquad E \vdash_{obj} T \qquad r \in \mathcal{H}.
\end{array}$$

The first of these is given, the second follows from the way $E \vdash_{method} \mathsf{m}\colon T \to U : R$ is derived, and the third follows from the way the given operational-semantics judgement is derived.

We now apply the induction hypothesis as planned, and get:

$$r, \sigma, r', \sigma', \emptyset \Vdash R \qquad (\sigma, \mu) \preceq (\sigma', \mu') \qquad E, \sigma', \mu' \Vdash r' : U \qquad E \Vdash \sigma', \mu'.$$

Now for the proof obligations:

- $$\begin{array}{ll}
 & r, \sigma, r', \sigma', S \Vdash \grave{r} \neq nil \;\Rightarrow\; R \\
= & \qquad \{ \;\; \text{definition of holds} \;\; \} \\
 & \vdash_{fol} (\grave{r} \neq nil \;\Rightarrow\; R)[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma'][S] \\
\Leftarrow & \qquad \{ \;\; \text{substitution is monotonic} \;\; \} \\
 & \vdash_{fol} R[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma'][S] \\
\Leftarrow & \qquad \{ \;\; \text{the substitution function } [S] \text{ is monotonic in the} \\
 & \qquad\quad \text{domain of } S \;\; \} \\
 & \vdash_{fol} R[\grave{r}, \grave{\sigma}, \acute{r}, \acute{\sigma} := r, \sigma, r', \sigma'][\emptyset] \\
= & \qquad \{ \;\; \text{definition of holds} \;\; \} \\
 & r, \sigma, r', \sigma', \emptyset \Vdash R
\end{array}$$

  which comes directly from the induction hypothesis.
- $(\sigma, \mu) \preceq (\sigma', \mu')$. Directly from induction hypothesis.
- $E, \sigma', \mu' \Vdash r' : U$. Directly from induction hypothesis.
- $E \Vdash \sigma', \mu'$. Directly from induction hypothesis.

And that completes all cases of the proof. □

References

ABADI, MARTÍN AND CARDELLI, LUCA 1996. *A Theory of Objects*. Springer-Verlag.

ABADI, MARTÍN AND LEINO, K. RUSTAN M. 1997. A logic of object-oriented programs. In *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, Volume 1214 of *Lecture Notes in Computer Science*. Springer, 682–696.

AMADIO, ROBERTO M. AND CARDELLI, LUCA 1993. Subtyping recursive types. *ACM TOPLAS 15*, 4 (Sept.), 575–631.

AMERICA, PIERRE AND DE BOER, FRANK 1994. Reasoning about Dynamically Evolving Process Structures. *Formal Aspects of Computing 6*, 3, 269–316.

CHAMBERS, CRAIG 1997. The Cecil Language: Specification & Rationale, version 2.1, March 7, 1997. Available from `http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html`.

DHARA, KRISHNA KISHORE AND LEAVENS, GARY T. 1997. Forcing behavioral subtyping through specification inheritance. Tech. Report TR #95-20c, Iowa State University, Department of Computer Science.

DIJKSTRA, EDSGER W. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

GOSLING, JAMES, JOY, BILL, AND STEELE, GUY 1996. *The Java™ Language Specification*. Addison-Wesley.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. of the ACM 12*, 10 (Oct.), 576–580,583.

LANO, KEVIN AND HAUGHTON, HOWARD 1994. *Object-Oriented Specification Case Studies*. Prentice Hall, New York.

LEAVENS, GARY TODD 1989. *Verifying Object-Oriented Programs that Use Subtypes*. PhD thesis, MIT Laboratory for Computer Science. Available as Technical Report MIT/LCS/TR-439.

LEINO, K. RUSTAN M. 1995. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology. Available as Technical Report Caltech-CS-TR-95-03.

LEINO, K. RUSTAN M. 1997. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*. Proceedings available from `http://www.cs.williams.edu/~kim/FOOL/FOOL4.html`.

LEINO, K. RUSTAN M. AND NELSON, GREG 1995. Object-oriented guarded commands. Internal manuscript KRML 50, Digital Equipment Corporation Systems Research Center.

NELSON, GREG 1983. Verifying Reachability Invariants of Linked Structures. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 38–47.

NELSON, GREG, EDITOR 1991. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ.

POETZSCH-HEFFTER, ARND AND MÜLLER, PETER 1998. Logical Foundations for Typed Object-Oriented Languages. In *Programming Concepts and Methods, PROCOMET '98*, IFIP. Chapman & Hall, 404–423.