Graph-Based Object-Oriented Hoare Logic*

Liang Zhao[1], Shuling Wang[2,**], and Zhiming Liu[3]

[1] Institute of Computing Theory and Technology, Xidian University, Xi'an, China
[2] State Key Lab. of Comput. Sci., Institute of Software,
Chinese Academy of Sciences, Beijing, China
[3] United Nations University - International Institute for Software Technology,
Macao SAR, China
wangsl@ios.ac.cn

Abstract. We are happy to contribute to this volume of essays in honor of He Jifeng on the occasion of his 70th birthday. This work combines and extends two recent pieces of work that He Jifeng has made significant contributions: the rCOS Relational Semantics of Object-Oriented Programs [4] and the Trace Model for Pointers and Objects [7]. It presents a graph-based Hoare Logic that deals with most general constructs of object-oriented (OO) programs such as assignment, object creation, local variable declaration and (possibly recursive) method invocation. The logic is built on a graph-based operational semantics of OO programs so that assertions are formalized as properties on graphs of execution states. We believe the logic is simple because 1) the use of graphs provides an intuitive visualization of states and executions of OO programs and thus it is helpful in thinking of and formulating clear specifications, 2) the logic follows almost the whole traditional Hoare Logic and the only exception is the backward substitution law which is not valid for OO programs, and 3) the mechanical implementation of the logic would not be much more difficult than traditional Hoare Logic. Despite the simplicity, the logic is powerful enough to reason about important OO properties such as aliasing and reachability.

Keywords: Hoare Logic, object-oriented program, state graph, aliasing.

1 Introduction

Correct design of an OO program from a specification is difficult. A main reason is that the execution states of an OO program are complex, due to the complex relation among the objects, aliasing, dynamic binding and polymorphism. This makes it hard to understand, to formulate and to reason about properties on behaviors of the program. Complexity is in general the cause of breakdowns of

* The research is supported by the NSFC Grant Nos. 61133001, 61103013, 91118007 and 61100061, Projects GAVES and PEARL funded by Macao Science and Technology Development Fund, and National Program on Key Basic Research Project (973 Program) Grant No.2010CB328102.

** Corresponding author.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): He Festschrift, LNCS 8051, pp. 374–393, 2013.
© Springer-Verlag Berlin Heidelberg 2013

a system and OO programs are typically prone to errors of a null pointer (or reference), an inaccessible object and aliases [7].

A formal semantic model must first contribute to *conceptual clarification* for better understanding so as to master the complexity better, and then help the thinking, formulating and reasoning about assertions of programs. To support the development of techniques and tools for analyzing and reasoning about programs, a logic is needed which should be defined based on the semantics. Obviously, a simple semantic model is essential for the definition of a logic that is easy to use for writing specifications and doing formal reasoning, and for implementing mechanical assistance.

In our earlier work [9], a graph-based operational semantics is defined and implemented for an OO programming language that is originally defined with a denotational semantics and a refinement calculus [4,18] for the rCOS method of component-based model-driven design [2,12]. In this semantics, objects of a class and execution states of a program are defined as directed and labeled graphs. A node represents an object or a simple datum. However, in the former case, the node is not labeled by an explicit reference value, but by the name of its runtime type that is a name of a class of the program. An edge is labeled by the name of a field of the source object referring to the target object. The advantage of the semantics lies in its naturalness in characterizing OO features, including the stack, heap, garbage, polymorphism and aliasing, and its intuitiveness for thinking and formulating properties of the execution of a program. Another good nature of the semantics is that it is *location independent*.

In this paper, we use the graph-based semantics to define a modest Hoare Logic for OO programs. There are mainly three OO features that make it difficult for Hoare Logic to be directly applied for specifying and reasoning about OO programs.

1. Side effects in assignment due to reference aliasing cause the invalidity of the (syntactic) backward substitution law

$$\{p[e/x]\}\, x := e\, \{p\}.$$

   For example, $\{(y.a = 4)[3/x.a]\}\, x.a := 3\, \{y.a = 4\}$ does not hold if x and y are *aliasing*, i.e. referring to the same object. There is a need of a rule for object creation which has side effects due to aliasing, too.
2. Dynamic method binding and recursions of methods make the specification of method invocations delicate.
3. Rules are needed for reasoning about dynamic typing.

In classical Hoare Logic [5], a *specification* or Hoare triple $\{p\}\, c\, \{q\}$ is defined in the way that p is a weakest precondition of the command c with respect to the postcondition q, e.g. the backward substitution law for assignment. However, the existence of aliasing makes it not so natural to propose a specification for OO commands in this way. Especially, it is very difficult to calculate a

precondition of an object creation given a postcondition that refers to the newly created object. This motivates us to take a *pre-to-post* approach, i.e. to calculate a postcondition from a precondition. Considering the example above, a correct specification should be

$$\{y.a = 4 \wedge x.a = V\}\, x.a := 3\, \{(y.a = 4)[V/x.a] \wedge x.a = 3\},$$

where we use a *logic variable* V to record the initial value of $x.a$. To deal with the problem of aliasing, we introduce a special substitution $[V/x.a]$ in the postcondition which intuitively means to substitute every term $e.a$ by V where e is an alias of x. Syntactically, $(y.a = 4)[V/x.a]$ is defined as $(V \triangleleft y = x \triangleright y.a) = 4$, which involves a *conditional term* $V \triangleleft y = x \triangleright y.a$. The meaning of the term is clear: it behaves as V if y is an alias of x, or as $y.a$ otherwise. If needed, we can further eliminate the auxiliary logic variable V and arrive at a more intuitive specification $\{y.a = 4\}\, x.a := 3\, \{(y.a = 3 \triangleleft y = x \triangleright y.a = 4) \wedge x.a = 3\}$. Notice that $y.a = 3$ is implied from $y = x$ and $x.a = 3$. This is actually due to a property of aliasing: if x and y are aliasing and a value 3 is reachable from x through a navigation path a, the value 3 is reachable from y through the same navigation path a. In general, aliasing terms are identical concerning *reachability*.

In our pre-to-post approach, the specification of an object creation is straightforward. Consider a precondition p and a command $C.\mathsf{new}(x.a)$ which creates an object of a class C and makes $x.a$ refer to the object. The specification can be of the form

$$\{p \wedge x.a = V\}\, C.\mathsf{new}(x.a)\, \{p[V/x.a] \wedge \exists U \cdot (x.a = U \wedge U : C \wedge q)\}.$$

Like in the specification of an assignment, we make use of the special substitution $[V/x.a]$ so that $p[V/x.a]$ holds in the postcondition for any precondition p. Besides, we use a fresh logic variable U to refer to the newly created object, thus U is reachable from x through the navigation path a, i.e. $x.a = U$, and the *runtime type* of U is C, i.e. $U : C$. The rest part q of the postcondition says attributes of U have been initialized to their default values and U can only be accessed through the navigation path a from x in this state.

The specification of a method invocation $e.m(x; y)$ is more delicate than that of an assignment or object creation. To realize the OO mechanism of dynamic method binding, we choose the method m according to the runtime type C of e, i.e. $e : C$, instead of the type of e declared. To deal with mutually recursive methods, we take the general approach that is to assume a set of specifications of method invocations and to prove the specification of bodies of these methods based on these assumptions, e.g. in [6,1,14]. However, the assumptions made in these work often rely on actual parameters of the method invocations, which makes the proof complicated as multiple assumptions with different parameters are needed for the invocation of one method. For simplicity and also efficiency, we introduce an auxiliary command $C :: m()$ which means the general execution of a method $C :: m$, i.e. m of class C. The auxiliary command enables the assumption of *method invariants* of the form $\{p\}\, C :: m()\, \{q\}$. Such an invariant is general and capable of deriving the specification of an invocation of $C :: m$

with any actual parameters. On the other hand, the invariant itself is free of actual parameters, so only one invariant is needed for each method.

As for the problem of typing, there are two solutions. The first is to define a type system along with the logic, and the second is, similar to Lamport's TLA [11], to state correct typing as assertion and provide the rules for type checking too. To keep the simplicity of the presentation, we leave the problem of typing out of this paper, but the type system defined with the graph-based operational semantics in [10] shows that either solution could work with the logic. Another restriction in this logic is that we do not deal with attribute shadowing.

The rest of the paper is organized as follows. Section 2 briefs our notations of graphs for OO programs. Sections 3 and 4 then present the underlying assertion language and the proof system, respectively. The soundness of the logic is discussed in Section 5. Finally, conclusions are drawn with discussions on related and future work.

2 Graph Representation of OO Programs

This section summarizes the graph notations of class structures and execution states of OO programs. Details can be found in our previous work [9,10].

2.1 An OO Language

We adopt the formal language of the rCOS method [4] as the basis of our discussion. It is a general OO language with essential OO features such as object creation, inheritance, dynamic method binding, and so on.

The language is equipped with a set of primitive data types, such as *Int* and *Bool*, and a set of built-in operations $f, \cdots$ on these types. Besides, let $C, D, \cdots$ range over classes $\mathcal{C}$; $S, T, \cdots$ range over types $\mathcal{T}$, including classes and data types; $a, x, y, \cdots$ range over attributes and variables $\mathcal{A}$; $m, \cdots$ range over methods $\mathcal{M}$; and $l, \cdots$ range over literals $\mathcal{L}$, including the null reference *null*. The syntax of the language is given in Fig. 1, where text occurring in square brackets is optional and overlined text $\overline{u}$ denotes a sequence $u_1, u_2, \ldots, u_k (k \geq 0)$.

A program *prog* is a sequence of class declarations *cdecls* followed by a main method *Main* which defines the execution of the whole program. A class C is declared optionally as a *direct subclass* of another class D, thus there is no multiple inheritance. An attribute declaration *adecl* consists of its type, name and default initial value, which is a literal. An attribute declared in a class cannot be re-declared in its subclasses, i.e., we do not consider attribute shadowing. A method declaration *mdecl* consists of the method name m, its value parameters $\overline{S}\ \overline{x}$, result parameters $\overline{T}\ \overline{y}$, and body command c. Notice that a method has result parameters instead of returning values directly. This is to make sure that expressions have no side effects. As a key feature of OO, a method is allowed to be overridden in a subclass, but its signature $m(\overline{S}; \overline{T})$ must be preserved.

program	$prog$	$::= cdecls \bullet Main$
class declarations	$cdecls$	$::= cdecl \mid cdecl; cdecls$
class declaration	$cdecl$	$::= \mathsf{class}\ C\ [\mathsf{extends}\ D]\ \{\overline{adecl}; \overline{mdecl}\}$
attribute definition	$adecl$	$::= T\ a = l$
method definition	$mdecl$	$::= m(\overline{S}\ \overline{x}; \overline{T}\ \overline{y})\{c\}$
command	c	$::= \mathsf{skip} \mid le := e \mid C.\mathsf{new}(le) \mid \mathsf{var}\ T\ x\, [= e] \mid \mathsf{end}\ x$ $\mid e.m(\overline{e}; \overline{le}) \mid C :: m() \mid c; c \mid c \triangleleft b \triangleright c \mid b * c$
expression	e	$::= le \mid \mathsf{self} \mid l \mid f(\overline{e})$
l-expression	le	$::= x \mid e.a$
boolean expression	b	$::= e \mid e = e \mid \neg b \mid b \wedge b$
main method	$Main$	$::= (\overline{ext}; c)$
external variable	ext	$::= T\ x = l$

Fig. 1. Syntax of rCOS language

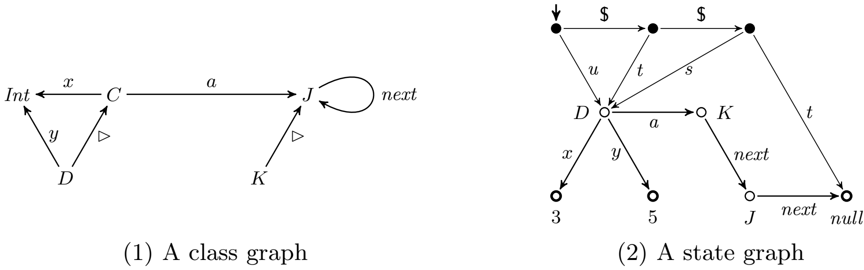
A command can be simply skip that does not do anything; $le := e$ that assigns e to le; $C.\mathsf{new}(le)$ that creates an object of class C and attaches it to le; var $T\ x = e$ that declares a local variable x of type T with initial value e, where e is by default the zero value $zero(T)$ of T; end x that ends the scope of x; or $e.m(\overline{e}; \overline{le})$ that invokes the method m of the object e refers to, with actual value parameters $\overline{e}$ and actual result parameters $\overline{le}$. Commands for sequential composition $c_1; c_2$, conditional choice $c_1 \triangleleft b \triangleright c_2$ and loop $b * c$ are also allowed. In addition, we introduce an auxiliary command $C :: m()$ to represent the general execution of the method $C :: m$, i.e. m defined in class C. Such a command will be used for the specification of *method invariants*.

Expressions include *assignable expressions* le, or simply l-expressions; the special self variable that represents the currently active object; literals l; and expressions $f(\overline{e})$ constructed with operations f of data types. Notice that expressions of the language have no side effects.

2.2 Class Graph and State Graph

The class declarations of a program can be represented as a directed and labeled graph, called a *class graph* [9]. In a class graph, a node represents a type T, which is either a class or a data type. There are two kinds of edges. An *attribute edge* $C \xrightarrow{a} T$, which is labeled by an attribute name a, represents that C has an attribute a of type T, while an *inheritance edge* $C \xrightarrow{\triangleright} D$, which is labeled by a designated symbol $\triangleright$, represents that C is a direct subclass of D. Notice that the source of an edge and the target of an inheritance edge must be nodes of classes. An example of class graph is shown in Fig. 2(1).

Given a class graph, we use $C \triangleright D$ to denote C is a direct subclass of D, and $\preccurlyeq$ the subclass relation that is the reflexive and transitive closure of $\triangleright$. We also use $Attr(C)$ to denote the set of attributes of C, including those inherited from C's superclasses. For an attribute $a \in Attr(C)$, we use $init(C, a)$ to denote its initial value. Besides, we introduce two partial functions $mtype(C :: m)$



(1) A class graph (2) A state graph

Fig. 2. Class graph and state graph

and $mbody(C :: m)$ for looking up the signature and the body of a method m of a class C, respectively.

$$mtype(C :: m) \mathrel{\widehat{=}} \begin{cases} (\overline{S}; \overline{T}) & \text{if } m(\overline{S}\ \overline{x}; \overline{T}\ \overline{y})\{c\} \text{ is defined in } C \\ mtype(D :: m) & \text{otherwise, if } C \rhd D \end{cases}$$

$$mbody(C :: m) \mathrel{\widehat{=}} \begin{cases} (\overline{x}; \overline{y}; c) & \text{if } m(\overline{S}\ \overline{x}; \overline{T}\ \overline{y})\{c\} \text{ is defined in } C \\ mbody(D :: m) & \text{otherwise, if } C \rhd D \end{cases}$$

With these functions, a class graph is used for type checking [10]. In addition, the class graph of an OO program is regarded as an abstract type whose instances are graphs representing executions states of the program, called *state graphs*.

Let $\mathcal{N}$ be an infinite set of node names and consider $\mathcal{A}^+ \mathrel{\widehat{=}} \mathcal{A} \cup \{\mathsf{self}, \$\}$ as the set of edge labels.

Definition 1 (State graph). *A* state graph *is a rooted, directed and labeled graph* $G = \langle N, E, \rho_{\mathsf{t}}, \rho_{\mathsf{v}}, r \rangle$, *where*

- $N \subseteq \mathcal{N}$ *is the set of nodes, denoted by* $G.node$,
- $E \subseteq N \times \mathcal{A}^+ \times N$ *is the set of edges, denoted by* $G.edge$,
- $\rho_{\mathsf{t}} : N \rightharpoonup \mathcal{C}$ *is a partial function from nodes to types, denoted by* $G.type$,
- $\rho_{\mathsf{v}} : N \rightharpoonup \mathcal{L}$ *is a partial function from nodes to values, denoted by* $G.value$,
- $r \in N$ *is the root of the graph, i.e. without incoming edges, denoted by* $G.root$,
- *starting from* r, *the* \$*-edges, if there are any, form a path such that except for* r *each node on the path has only one incoming edge.*

A state graph is a snapshot of the state at one time of the program execution, consisting of the existing objects, their attributes, as well as variables of different scopes that refer to these objects. Specifically, a state graph G has three kinds of nodes: *object nodes*, *value nodes* and *scope nodes*, representing objects, values and scopes, respectively. Object nodes are the domain of $G.type$ so that each object node is labeled by its class with outgoing edges representing its attributes. Value nodes are the domain of $G.value$ so that each value node is labeled by a value. We assume a value node is in the state graph when needed, as otherwise it can always be added. A scope node has outgoing edges representing variables declared in the scope. In addition, scope nodes are associated with a \$-labeled path and they constitute the *stack* of the state graph. The first (scope) node of

the stack, i.e. the source of the \$-labeled path, represents the scope of the current execution. It is the *root* node of the graph through which variables and objects of the state can be accessed. An example state graph is shown in Fig. 2(2).

To represent a sound state, a state graph G should satisfy a few conditions of *well-formedness* [9], e.g. outgoing edges of each node have distinct labels. In addition, a state graph should be *correctly typed* with respect to the class graph of a program. Intuitively, the class of each object node is defined in the class graph and each attribute is correctly typed according to the class graph. For example, the state graph in Fig. 2(2) is correctly typed with respect to the class graph in Fig. 2(1).

Trace and evaluation. We use the term *trace*, or navigation path, to denote a sequence of edge labels. In a state graph, every path $G.root \xrightarrow{x_1} n_1 \xrightarrow{x_2} \cdots \xrightarrow{x_k} n_k$ from the root is uniquely determined by its trace $x_1.x_2.\cdots.x_k$. We thus allow the interchange between a root-originating path and its trace. Besides, we do not distinguish state graphs different only in the choice of their node names, and this is formalized by the notion of graph isomorphism [9]. Notice that isomorphic state graphs have the same set of traces.

Given a state graph G that represents a state, the evaluation of an expression e returns its value $eval(e)$ and runtime type $rtype(e)$ in the state [9]. For most expressions e, the evaluation is simply the calculation of their traces $trace(e)$. If the trace of e targets at an object node o, $eval(e) \widehat{=} o$ and $rtype(e) \widehat{=} G.type(o)$. Otherwise, $eval(e) \widehat{=} G.value(v)$ which is a literal and $rtype(e) \widehat{=} \mathbf{T}(G.value(v))$. Here, $\mathbf{T}(l)$ denotes the type of a literal l. Notice that the trace of an expression e may not exist. In this case, the evaluation fails and we denote both $eval(e)$ and $rtype(e)$ as $\perp$. To sum up, every expression evaluates to an element in $\mathcal{V} \widehat{=} \mathcal{N} \cup \mathcal{L} \cup \{\perp\}$. Thus we call $\mathcal{V}$ the *value space.*

2.3 Graph Operation

We defined an operational semantics of the OO language in terms of transitions $con \to con$ between *configurations* [9]. Here, a configuration con is either $\langle c, G\rangle$ representing a command c to be executed and a state G, or G representing the state that the execution terminates at. The semantics is simple in the sense that it is defined by a few basic operations on state graphs.

Swing. The most frequent operation on a state graph is an edge swing. Specifically, for an edge $d = v_1 \xrightarrow{a} v_2$ and a node v of G, $swing(G, d, v)$ is the graph obtained from G by making d target at v (instead of v_2). The swing $swing(G, \alpha, v)$ of a trace α is the swing of its last edge, see Fig. 3. The swing operation is used to define the semantics of an assignment: $\langle le := e, G\rangle \to swing(G, trace(le), eval(e))$.

New. Given a state graph G, a class C and a trace α in G, the operation $new(G, C, \alpha)$ creates an object node of class C with attributes initialized by default values and then swings α to the new object node, see Fig. 4. This operation is used to define the semantics of object creation: $\langle C.\mathsf{new}(le), G\rangle \to new(G, C, trace(le))$.

Push and pop. Let G be a state graph, x a variable and v a node of G. The operation $push(G, x, v)$ adds a new scope node r, with an outgoing edge labeled

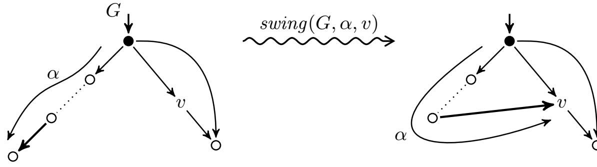


Fig. 3. Trace swing

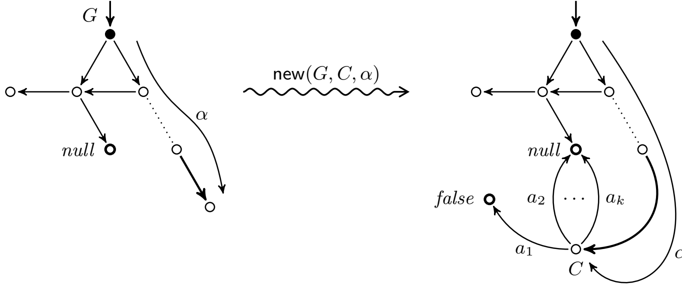


Fig. 4. Object creation

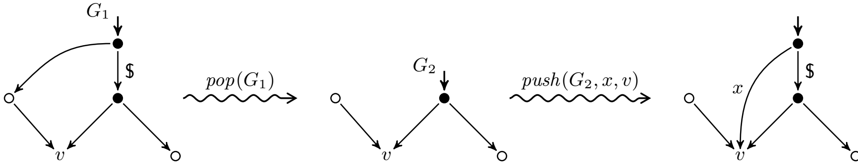


Fig. 5. Stack push and pop

by x and targeting at v, to the top of the stack so that r becomes the root of the result graph. In contrast, the operation $pop(G)$ removes the root node together with its outgoing edges from the graph, while the next scope node becomes the root. They are shown in Fig. 5. The push operation is used to define the declaration of a local variable: $\langle \mathsf{var}\ T\ x = e, G\rangle \to push(G, x, eval(e))$, as well as the switch of the execution into the method body at the beginning of a method invocation. Correspondingly, the pop operation is used to define the un-declaration of a local variable: $\langle \mathsf{end}\ x, G\rangle \to pop(G)$, as well as the switch of the execution out of the method body at the end of a method invocation.

3 Assertion Language

The advantage of our graph notations lies in both the intuitive understanding and the theoretical maturity of graphs. They are thus helpful to formulate clear and precise assertions on the execution of OO programs. In this section, we propose an assertion language as the basis of our Hoare Logic. It is a first-order language with equality characterizing the aliasing property.

$$\begin{array}{lrl} \text{assertion} & p ::= & P(\bar{t}) \mid t = t \mid t \uparrow \ \mid t : C \\ & & \mid \mathit{true} \mid \mathit{false} \mid \neg p \mid p \wedge p \mid \exists U \cdot p \\ \text{term} & t ::= & x \mid t.a \mid \mathsf{self} \mid l \mid f(\bar{t}) \\ & & \mid U \mid t \lhd t = t \rhd t \end{array}$$

Fig. 6. Syntax of the assertion language

Let $\mathcal{O}$ be the vocabulary of *logic variables* $U, V, \cdots$ and let $P, \ldots$ range over predicates. The syntax of the assertion language is given in Fig. 6. Assertions include $P(\bar{t})$ that applies a k-ary predicate P on a sequence of k terms $\bar{t}$; $t_1 = t_2$ that says terms t_1 and t_2 are *aliasing*; $t \uparrow$ that claims t successfully evaluates to a value not $\perp$; and $t : C$ that asserts the runtime type of t is class type C. General constructs of a first-order language are also allowed, such as negation $\neg p$, conjunction $p_1 \wedge p_2$ and (existential) quantification $\exists U \cdot p$. We regard $p_1 \Rightarrow p_2$, $p_1 \vee p_2$, $p_1 \Leftrightarrow p_2$, $\forall U \cdot p$ as shorthands for $\neg(p_1 \wedge \neg p_2)$, $\neg p_1 \Rightarrow p_2$, $(p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$, $\neg \exists U \cdot \neg p$, respectively.

The syntax of terms is simply an extension of that of expressions (see Fig. 1) with logic variables and conditional terms. A logic variable U is introduced to record a constant value. This value cannot be changed by the execution of commands since U never occurs in a command. A conditional term $t_1 \lhd t = t' \rhd t_2$ behaves as t_1 or t_2, depending on whether t and t' are aliasing or not. We use $lv(t)$ to denote the set of logic variables that occur in a term t, and $flv(p)$ the set of logic variables that occur *free*, i.e. not bound by quantifiers, in an assertion p.

3.1 Satisfaction of Assertion

As for the semantics of the assertion language, we characterize whether an assertion p is satisfied by a state graph G. Since assertions contain logic variables, we extend the notion of state graph with logic variables correspondingly.

Definition 2 (Extended state graph). *An* extended state graph *is a rooted, directed and labeled graph* $G = \langle N, E, \rho_{\mathsf{c}}, \rho_{\mathsf{v}}, r, v \rangle$*, where*

- N, ρ_{c}, ρ_{v} *and* r *are defined as in Definition 1,*
- $E \subseteq N \times (\mathcal{A}^+ \cup \mathcal{O}) \times N$ *is the set of edges, denoted by* $G.edge$*,*
- $v \in N$ *is a special node without incoming edges, denoted by* $G.lvar$*, such that each outgoing edge of* v *is labeled by a logic variable, and furthermore, each edge labeled by a logic variable is an outgoing edge of* v.

The extension to an original state graph G is mainly a special node $G.lvar$ with outgoing edges $G.lvar \xrightarrow{U_1} n_1, \ldots, G.lvar \xrightarrow{U_k} n_k$ recording a set of logic variables. We use $lv(G)$ to denote the set of logic variables $\{U_1, \ldots, U_k\}$ of G.

Notice that all the graph operations provided in Section 2.3, and thus the operational semantics, are applicable to extended state graphs. In fact, the execution of a command never changes the existence and values of logic variables. In the rest of the paper, a state graph always means an extended one.

Given a term t and a state graph G with $lv(t) \subseteq lv(G)$, the evaluation of t calculates the value $eval(t)$ and the runtime type $rtype(t)$ of t in G. For a logic

variable $U \in lv(G)$, there must be an edge $G.lvar \xrightarrow{U} n$ in G. If n is an object node, $eval(U) \mathrel{\widehat{=}} n$ and $rtype(U) \mathrel{\widehat{=}} G.type(n)$. If n is a value node, $eval(U) \mathrel{\widehat{=}} G.value(n)$ and $rtype(U) \mathrel{\widehat{=}} \mathbf{T}(G.value(n))$. The evaluation of a conditional term $t \equiv t_1 \triangleleft t' = t'' \triangleright t_2$ is the same as that of t_1 or t_2, depending on whether $eval(t') = eval(t'')$ or not. Other constructs of terms evaluate in the same way as those of expressions.

To reason about the satisfaction of predicates, we consider the notion of *interpretation*. An interpretation I of the assertion language interprets every k-ary predicate P as a k-ary relation on the value space $\mathcal{V}$, i.e. $I(P) \subseteq \mathcal{V}^k$. To calculate the satisfaction of a quantified assertion $\exists U \cdot p$, we introduce an operation that adds a logic variable U into a state graph G and makes it refer to a node n of G:

$$addv(G, U, n) \mathrel{\widehat{=}} G' \qquad \text{provided } U \notin lv(G),$$

where G' is the same as G except that $G'.edge = G.edge \cup \{G.lvar \xrightarrow{U} n\}$. Intuitively, $\exists U \cdot p$ is satisfied by G if there is an object node or value node n of G such that p is satisfied by $addv(G, U, n)$. The satisfaction of other assertions can be defined straightforwardly. For example, $t_1 = t_2$ is satisfied if t_1 and t_2 evaluate to the same value, $t\uparrow$ is satisfied if t evaluates to a value other than $\bot$, while $t : C$ is satisfied if the runtime type of t is C.

Definition 3 (Satisfaction of assertion). *For an assertion p, an interpretation I and a state graph G with $flv(p) \subseteq lv(G)$, we use $G \models_I p$ to denote that p is* satisfied *by G under I. It is defined inductively on the structure of p.*

- *For $p \equiv P(t_1, \ldots, t_k)$, $G \models_I p$ if $(eval(t_1), \ldots, eval(t_k)) \in I(P)$.*
- *For $p \equiv t_1 = t_2$, $G \models_I p$ if $eval(t_1) = eval(t_2)$.*
- *For $p \equiv t\uparrow$, $G \models_I p$ if $eval(t) \neq \bot$.*
- *For $p \equiv t : C$, $G \models_I p$ if $rtype(t) = C$.*
- *For $p \equiv true$, $G \models_I p$ always holds; for $p \equiv false$, $G \models_I p$ never holds.*
- *For $p \equiv \neg p_1$, $G \models_I p$ if $G \models_I p_1$ does not hold.*
- *For $p \equiv p_1 \wedge p_2$, $G \models_I p$ if $G \models_I p_1$ and $G \models_I p_2$,*
- *For $p \equiv \exists U \cdot p_1$, assume $U \notin lv(G)$ as U can be renamed by alpha-conversion. $G \models_I p$ if $addv(G, U, n) \models_I p_1$ for some object node or value node n of G.*

We say p is true *under I, denoted as $\models_I p$, if $G \models_I p$ for any state graph G with $flv(p) \subseteq lv(G)$. In addition, we say p is* valid, *denoted as $\models p$, if $\models_I p$ under any interpretation I.*

Notice that we use $\equiv$ to denote the equivalence in syntax. For example, $p \equiv t : C$ means p and $t : C$ represent the same syntactic assertion, i.e., p is exactly $t : C$.

It is straightforward to verify that the satisfaction of an assertion does not rely on the naming of nodes of the underlying state graph, i.e., $G \models_I p$ if and only if $G' \models_I p$ for isomorphic state graphs G and G'. This indicates that our OO assertion model, and further the proof system, is *location independent*.

The semantics of assertions provided in the above definition is consistent with the semantics of first-order logic. As a result, every valid formula of first-order

logic, e.g. $\neg\neg p \Leftrightarrow p$, is a valid assertion. In addition, it is straightforward to prove the validity of the following OO assertions, where $t \neq t'$ is a shorthand for $\neg t = t'$.

1. $t = t' \Rightarrow t_1 = t'_1$, provided t'_1 is obtained from t_1 by replacing one or more occurrence of t by t'. This assertion says that aliasing terms share the same properties.
2. $t.a \uparrow \Rightarrow t \uparrow \wedge\, t \neq \mathit{null}$. This assertion says that the evaluation of $t.a$ successes only if t evaluates to a non-*null* object.
3. $t : C \Rightarrow t \uparrow \wedge\, t \neq \mathit{null}$. This assertion says that only objects can have runtime class types.
4. $t \uparrow \Leftrightarrow \exists U \cdot U = t$ provided U is fresh. This assertion reflects the intuition that a logic variable is used to record a value.

4 Proof System

The assertion language enables us to define program *specifications*. A specification takes the form $\{p\}\, c\, \{q\}$, where p, q are assertions and c is a command. We call p and q the *precondition* and *postcondition* of the specification, respectively. Intuitively, such a specification means that if p holds before c executes, and when the execution of c terminates, then q holds after the execution. We will formally define the semantics of specifications in the next section. For a specification $\{p\}\, c\, \{q\}$, we always assume $\mathit{flv}(q) \subseteq \mathit{flv}(p)$. In fact, a specification that generates new free logic variables in the postcondition is not necessary. For example, by $\{x = y\}\, \mathsf{skip}\, \{x = V \wedge V = y\}$, we actually mean $\{x = y\}\, \mathsf{skip}\, \{\exists V \cdot x = V \wedge V = y\}$.

In this section, we present the Hoare proof system that consists of a set of *logic rules* for specifications of all constructs of the OO language presented in Section 2.1. Each logic rule defines a one-step proof (or derivation) of a *conclusion* from zero or more *hypotheses*. The conclusion takes the form of a specification, while each hypothesis can be either an assertion, a specification, or a *specification sequent* $\Phi \vdash \Psi$, where Φ and Ψ are sets of specifications. The specification sequent means Ψ can be proved (or derived) from Φ by applying the logic rules.

For natural specification of commands such as assignment and object creation, we define the proof system in a *pre-to-post* way. That is, each logic rule calculates the postcondition of a specification from an arbitrary precondition.

4.1 Assignment

For specification of an assignment $le := e$, we introduce two logic variables V_0 and V to record the values of le and e before the assignment, respectively. The l-expression le can be either a variable x or a navigation expression $e_0.a$.

In the former case $le \equiv x$, the specification is straightforward and $x = V$ holds in the postcondition.

$$\{p \wedge x = V_0 \wedge e = V\}\ x := e\ \{p[V_0/x] \wedge x = V\} \tag{1}$$

If the precondition p does not contain x, p also holds in the postcondition because the assignment only modifies the value of x. Otherwise, we can replace each x in p by its original value V_0, so that $p[V_0/x]$ holds in the postcondition.

In the latter case $le \equiv e_0.a$, we use an extra logic variable U to record the value of e_0, so that $U.a = V$ holds in the postcondition.

$$\{p \wedge e_0 = U \wedge U.a = V_0 \wedge e = V\}\ e_0.a := e\ \{p[V_0/U.a] \wedge U.a = V\} \tag{2}$$

For the rest of the postcondition, we introduce a special substitution $[V/U.a]$. Intuitively, $p[V/U.a]$ is obtained from p by replacing every (sub-)term of the form $t'.a$, where t' is an alias of U, by the logic variable V. As a result, the satisfaction of $p[V/U.a]$ is not compromised by the assignment. This substitution is formally defined according to the structure of assertions p, as well as terms t.

$$p[V/U.a] \mathrel{\widehat{=}} \begin{cases} P(t_1[V/U.a], \ldots, t_k[V/U.a]) & \text{if } p \equiv P(t_1, \ldots, t_k) \\ t_1[V/U.a] = t_2[V/U.a] & \text{if } p \equiv t_1 = t_2 \\ t[V/U.a] \uparrow & \text{if } p \equiv t \uparrow \\ t[V/U.a] : C & \text{if } p \equiv t : C \\ p & \text{if } p \equiv \mathit{true} \text{ or } \mathit{false} \\ \neg p_1[V/U.a] & \text{if } p \equiv \neg p_1 \\ p_1[V/U.a] \wedge p_2[V/U.a] & \text{if } p \equiv p_1 \wedge p_2 \\ \exists V' \cdot p_1[V/U.a] & \text{if } p \equiv \exists V' \cdot p_1 \text{ where } V' \text{ is not } U \text{ or } V \end{cases}$$

$$t[V/U.a] \mathrel{\widehat{=}} \begin{cases} t & \text{if } t \equiv x, \mathsf{self}, l \text{ or } V \\ t_1[V/U.a].a_1 & \text{if } t \equiv t_1.a_1,\ a_1 \not\equiv a \\ V \lhd t_1[V/U.a] = U \rhd t_1[V/U.a].a & \text{if } t \equiv t_1.a \\ f(t_1[V/U.a], \ldots, t_k[V/U.a]) & \text{if } t \equiv f(t_1, \ldots, t_k) \\ t_1[V/U.a] \lhd t'[V/U.a] = t''[V/U.a] \rhd t_2[V/U.a] & \text{if } t \equiv t_1 \lhd t' = t'' \rhd t_2 \end{cases}$$

4.2 Object Creation

For specification of an object creation $C.\mathsf{new}(le)$, we use a logic variables V_0 to record the original value of le. Like in the specification of assignment, we need to consider two cases of le: a variable x, or a navigation expression $e_0.a$.

For $le \equiv x$, the logic rule is given as follows.

$$\begin{array}{l} \text{provided } V \text{ is fresh} \\ \{p \wedge x = V_0\}\ C.\mathsf{new}(x) \\ \{p[V_0/x] \wedge \exists V \cdot x = V \wedge V : C \wedge V = C_{init} \wedge V \neq p[V_0/x]\} \end{array} \tag{3}$$

Similar to Rule (1), $p[V_0/x]$ holds in the postcondition given any precondition p. In addition, we introduce a fresh logic variable V, which is existentially quantified, in the postcondition to record the reference to the new object of class C, thus $x = V$ and $V : C$ hold. For the rest of the postcondition, $V = C_{init}$ says that the attributes of the new object are initialized, while $V \neq p[V_0/x]$ indicates that the new object can only be accessed from x but not $p[V_0/x]$.

Formally, $V = C_{init}$ is a shorthand for $V.a_1 = init(C, a_1) \wedge \ldots \wedge V.a_k = init(C, a_k)$, provided $Attr(C) = \{a_1, \ldots, a_k\}$. $V \neq p$ is a shorthand for $V \neq t_1 \wedge \ldots \wedge V \neq t_k$, where $t_1, \ldots, t_k$ are the *free maximum* terms occurring in p. A term is free if it does not contain a quantified logic variable, while a term is maximum if it does not occur as a sub-term of another term.

For $le \equiv e_0.a$, we use a logic variable U to record the original value of e_0.

$$\begin{array}{l}\text{provided } V \text{ is fresh}\\ \{p \wedge e_0 = U \wedge U.a = V_0\}\, C.\mathsf{new}(e_0.a)\\ \{p[V_0/U.a] \wedge \exists V \cdot U.a = V \wedge V : C \wedge V = C_{init} \wedge V \neq p[V_0/U.a]\}\end{array} \tag{4}$$

This rule is similar to Rule (3), while $p[V_0/U.a]$ holds in the postcondition.

4.3 Local Variable Declaration

We only consider the specification of $\mathsf{var}\ T\ x = e; c; \mathsf{end}\ x$ where x is initialized by e, because $\mathsf{var}\ T\ x; c; \mathsf{end}\ x$ is a shorthand for $\mathsf{var}\ T\ x = zero(T); c; \mathsf{end}\ x$. We use a logic variable V to record the original value of e, so that the specification of $\mathsf{var}\ T\ x = e; c; \mathsf{end}\ x$ depends on a specification of c with $x = V$ in the precondition.

If x does not occur in a precondition p or the expression e, we can simply use p as a precondition of c which will lead to a postcondition q. To obtain the overall postcondition that holds after the execution of $\mathsf{end}\ x$, we hide all occurrences of x in q by existential quantification.

$$\text{provided } U \text{ is fresh}\quad \frac{\{p \wedge x = V\}\, c\, \{q\}}{\{p \wedge e = V\}\, \mathsf{var}\ T\ x = e; c; \mathsf{end}\ x\, \{\exists U \cdot q[U/x]\}}$$

If x occurs in p or e, we need to record the value of x by a logic variable W so as to recover x after the execution of $\mathsf{var}\ T\ x = e; c; \mathsf{end}\ x$. Of course, p cannot be used as a precondition of c until we replace each occurrence of x by W.

$$\text{provided } U \text{ is fresh}\quad \frac{\{p[W/x] \wedge x = V\}\, c\, \{q\}}{\{p \wedge e = V \wedge x = W\}\, \mathsf{var}\ T\ x = e; c; \mathsf{end}\ x\, \{(\exists U \cdot q[U/x]) \wedge x = W\}}$$

For conciseness, we unify the above two cases into a single logic rule.

$$\text{provided } U \text{ is fresh; let } p_* \text{ be } p \wedge e = V\quad \frac{\{p[W/x] \wedge x = V\}\, c\, \{q\}}{\{p_* \wedge (?p_*)x = W\}\, \mathsf{var}\ T\ x = e; c; \mathsf{end}\ x\, \{(\exists U \cdot q[U/x]) \wedge (?p_*)x = W\}} \tag{5}$$

Here, $(?p)w = t$ is a designated assertion defined as follows, in which w is either a variable or self. We will also use it in the specification of method invocations.

$$(?p)w = t \;\widehat{=}\; \begin{cases} w = t & \text{if } w \text{ occurs in } p \\ true & \text{otherwise} \end{cases}$$

4.4 Method Invocation

A key feature of OO programs is the dynamic binding of method invocation. That is, a method invocation $e.m(ve; re)$ is an invocation of the method $C :: m$ where C is the runtime type of e. In our logic, the condition "C is the runtime type of e" is naturally characterized by an assertion $e : C$.

For specification of an invocation of $C :: m$, we make use of a *method invariant* $\{p\}\, C :: m()\, \{q\}$ that is a specification of the general execution of the method $C :: m()$. Specifically, the semantics of $C :: m()$ is defined the same as that of the body command of $C :: m$. Therefore, if a method $C :: m$ is non-recursive, its invariant is directly proved from the specification of its body command.

$$\begin{array}{l}\text{provided } \mathit{mbody}(C :: m) = (x; y; c) \\ \dfrac{\{p\}\, c\, \{q\}}{\{p\}\, C :: m()\, \{q\}}\end{array} \tag{6}$$

Once a method invariant is proved, it is used to derive specifications of invocations $e.m(ve; re)$ of the method $C :: m$ with any (well-typed) actual parameters $(ve; re)$, where $e : C$.

$$\begin{array}{l}\text{provided } \mathit{mtype}(C{::}m) = (S; T);\ \mathit{mbody}(C{::}m) = (x; y; c);\ W_4, W_5, W_6, W_7 \text{ fresh;} \\ \text{let } p_* \text{ be } p \wedge e = U \wedge ve = V \wedge re \uparrow \\ \dfrac{\{p[W_1, W_2, W_3/\mathsf{self}, x, y] \wedge \mathsf{self} = U \wedge x = V \wedge y = \mathit{zero}(T)\}\, C :: m()\, \{q\}}{\begin{array}{c}\{p_* \wedge U : C \wedge \underline{re}? = V_0 \wedge (?p_*)\mathsf{self} = W_1 \wedge (?p_*)x = W_2 \wedge (?p_*)y = W_3\} \\ e.m(ve; re)\, \{(\exists W_4, W_5, W_6, W_7 \cdot q[W_4, W_5, W_6, W_7/\mathsf{self}, x, y, re[V_0/_?]] \\ \wedge re[V_0/_?] = W_6) \wedge (?p_*)\mathsf{self} = W_1 \wedge (?p_*)x = W_2 \wedge (?p_*)y = W_3\}\end{array}}\end{array} \tag{7}$$

Given a precondition p of the method invocation, we use logic variables U and V to record respectively the values of e and the value parameter ve in p, so that U and V are respectively the values of self and x in the precondition of $C :: m()$. Besides, the result parameter re must have a value to receive the result of the invocation, thus $p_* \equiv p \wedge e = U \wedge ve = V \wedge re \uparrow$ is part of the precondition of the invocation. If p_* contains self, x and y, we record their values by logic variables W_1, W_2 and W_3, respectively, and recover them after the invocation. However, p_* may not contain all of them. In the case p_* contains self and y but not x, for example, we only need to introduce the corresponding logic variables W_1 and W_3. To unify different cases, we make use of the notation $(?p_*)w = t$ defined in Section 4.3. This is similar to Rule (5) for local variable declaration. The rest of the postcondition is obtained from that of $C :: m()$ by hiding self, x and y that are local to $C :: m()$. For this, we first replace them by fresh logic variables W_4, W_5 and W_6, respectively, thus W_6 actually records the result of the invocation. Then, we hide these logic variables with existential quantification.

The rest of the rule is the return of result of the invocation W_6 to the result parameter re. There are two cases of re: a variable x or a navigation expression $e_0.a$. In the former case, we simply return W_6 to x. In the latter case, we introduce an extra logic variable V_0 to record the *parent object* e_0 of re before the invocation and return W_6 to $V_0.a$ after the invocation. This is the so-called *early binding*

of result parameters. For unification of the two cases, we introduce a designated assertion $\underline{le}? = V$ and a designated term $le[V/_?]$ for l-expressions le and logic variables V.

$$\underline{le}? = V \mathrel{\widehat{=}} \begin{cases} e = V & \text{if } le \equiv e.a \\ true & \text{if } le \equiv x \end{cases} \qquad le[V/_?] \mathrel{\widehat{=}} \begin{cases} V.a & \text{if } le \equiv e.a \\ x & \text{if } le \equiv x \end{cases}$$

Of course, we need to hide the value of $re[V_0/_?]$ before returning W_6 to $re[V_0/_?]$.

To avoid unnecessary name conflicts, we always assume that the result parameter re of a method invocation $e.m(ve; re)$ has a different name from a formal parameter x of a method declaration. Otherwise, e.g. $re \equiv x$, we can replace $e.m(ve; x)$ by an equivalent command $\mathsf{var}\ T\ z; e.m(ve; z); x := z; \mathsf{end}\ z$, where T is the type of the result parameter of m and z is a fresh name.

Recursive method invocation. Rule (6) is not strong enough to prove the invariants of recursive methods. For example, if the body c of a method $C :: m$ involves an invocation of $C :: m$ itself, the hypothesis $\{\ldots\}\, c\, \{\ldots\}$ of the rule would in turn rely on the conclusion $\{\ldots\}\, C :: m()\, \{\ldots\}$ of the rule and thus cannot be proved.

We take the general approach to dealing with recursion [6,1]. Assume a group of mutually recursive methods $C_1 :: m_1, \ldots, C_k :: m_k$, each of which may call the whole group in its body. If the specifications of the method bodies $\{p_1\}\, c_1\, \{q_1\}; \ldots; \{p_k\}\, c_k\, \{q_k\}$ can be proved under assumptions of the invariants $\{p_1\}\, C_1 :: m_1()\, \{q_1\}; \ldots; \{p_k\}\, C_k :: m_k()\, \{q_k\}$, these assumptions are established.

$$\frac{\begin{array}{c}\text{provided } mbody(C_i :: m_i) = (x_i; y_i; c_i) \text{ for } i = 1, \ldots, k \\ \{p_i\}\, C_i :: m_i()\, \{q_i\}_{\,i=1,\ldots,k} \vdash \{p_i\}\, c_i\, \{q_i\}_{\,i=1,\ldots,k}\end{array}}{\{p_j\}\, C_j :: m_j()\, \{q_j\}_{\,1 \le j \le k}} \tag{8}$$

4.5 Other Constructs

Logic rules of sequential composition, conditional choice and while loop simply follow the traditional Hoare Logic [5].

$$\frac{\{p\}\, c_1\, \{p_1\} \qquad \{p_1\}\, c_2\, \{q\}}{\{p\}\, c_1; c_2\, \{q\}} \tag{9}$$

$$\frac{\{p \wedge b\}\, c_1\, \{q\} \qquad \{p \wedge \neg b\}\, c_2\, \{q\}}{\{p\}\, c_1 \lhd b \rhd c_2\, \{q\}} \tag{10}$$

$$\frac{\{p \wedge b\}\, c\, \{p\}}{\{p\}\, b * c\, \{p \wedge \neg b\}} \tag{11}$$

4.6 Auxiliary Rules

Besides rules for deriving specifications of various OO constructs, we have *auxiliary rules* that are useful to transform the precondition and postcondition of

a specification. First, we can make the precondition of a specification "stronger" and the postcondition "weaker". And this is the so-called *consequence rule.*

$$\text{provided } \mathit{flv}(p) \subseteq \mathit{flv}(p');\ \mathit{flv}(q') \subseteq \mathit{flv}(q)$$
$$\frac{p' \Rightarrow p \quad \{p\}\, c\, \{q\} \quad q \Rightarrow q'}{\{p'\}\, c\, \{q'\}} \tag{12}$$

Another rule is about *constant* assertions. An assertion p is called constant if it does not contain any variable x, self or navigation expression $e.a$. The satisfaction of a constant assertion cannot be changed by the execution of commands.

$$\text{provided } p \text{ is constant}$$
$$\{p\}\, c\, \{p\} \tag{13}$$

In addition, we can combine specifications of the same command by conjunction. We can also hide, by existential quantification, and rename logic variables.

$$\frac{\{p_1\}\, c\, \{q_1\} \qquad \{p_2\}\, c\, \{q_2\}}{\{p_1 \wedge p_2\}\, c\, \{q_1 \wedge q_2\}} \tag{14}$$

$$\frac{\{p\}\, c\, \{q\}}{\{\exists U \cdot p\}\, c\, \{\exists U \cdot q\}} \tag{15}$$

$$\text{provided } U \text{ never occurs in the scope of } \exists V$$
$$\frac{\{p\}\, c\, \{q\}}{\{p[V/U]\}\, c\, \{q[V/U]\}} \tag{16}$$

4.7 Example

We use an example to show the application of the proof system. Consider the following class declaration, with a recursive method $fact(Int\ x; Int\ y)$ to calculate the factorial of x and to return it to y.

```
class C{ ...;
    fact(Int x; Int y){
        (var Int z; self.fact(x − 1; z); y := z ∗ x; end z) ◁ x > 1 ▷ y := 1
    }
}
```

We are going to prove the method is correctly defined:

$$\{e : C \wedge z \uparrow\}\, e.fact(5; z)\, \{z = 5!\}.$$

For this, we need to prove an invariant (INV) of the method:

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y \uparrow\}\, C :: fact()\, \{x = V \wedge V \geq 0 \wedge y = V!\}.$$

This invariant is strong enough to derive the above conclusion using Rule (7) of method invocation, as well as auxiliary rules (12), (13), (14) and (15).

We use Rule (8) of recursion to prove (INV). That is, assuming (INV), we prove the following specification of the method body, denoted as (BOD).

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y\uparrow\}$$
$$c_1 \triangleleft x > 1 \triangleright y := 1\, \{x = V \wedge V \geq 0 \wedge y = V!\}$$

where c_1 is $\mathsf{var}\ \mathit{Int}\ z; \mathsf{self}.\mathit{fact}(x-1; z); y := z * x; \mathsf{end}\ z$.

From (INV) and Rule (7) of method invocation, as well as auxiliary rules (12), (13), (14), (15) and (16), we have

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y\uparrow \wedge\, x > 1 \wedge z = V_0\}$$
$$\mathsf{self}.\mathit{fact}(x-1; z)\, \{V \geq 1 \wedge z = (V-1)! \wedge x = V \wedge y\uparrow\}.$$

From Rule (1) of assignment, as well as Rule (15), we have

$$\{V \geq 1 \wedge z = (V-1)! \wedge x = V \wedge y\uparrow\}$$
$$y := z * x\, \{x = V \wedge V \geq 1 \wedge y = V! \wedge z = (V-1)!\}.$$

By Rule (9) of sequential composition, the above two specifications lead to

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y\uparrow \wedge\, x > 1 \wedge z = V_0\}$$
$$\mathsf{self}.\mathit{fact}(x-1; z); y := z * x\, \{x = V \wedge V \geq 1 \wedge y = V! \wedge z = (V-1)!\}.$$

Then, using Rule (5) of local variable declaration, as well as auxiliary rules (12) and (15), we arrive at the following specification.

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y\uparrow \wedge\, x > 1\}\, c_1 \{x = V \wedge V \geq 0 \wedge y = V!\}$$

On the other hand, we use Rule (1) of assignment, as well as auxiliary rules (12) and (15), and arrive at the following specification.

$$\{x = V \wedge V \geq 0 \wedge \mathsf{self} = U \wedge U : C \wedge y\uparrow \wedge \neg(x > 1)\}\, y := 1 \{x = V \wedge V \geq 0 \wedge y = V!\}$$

Finally, (BOD) is proved from the above two specifications using Rule (10) of conditional choice.

5 Soundness of the Logic

We have provided a Hoare proof system for specification of OO programs, denoted as $\mathcal{H}$, which consists of a set of logic rules (1) to (16). In this section, we show that $\mathcal{H}$ is sound. For this, we need to define the semantics of specifications.

Let $\{p\}\, c\, \{q\}$ be a specification and I be an interpretation of assertions. We say $\{p\}\, c\, \{q\}$ is *true* under I, denoted as $\models_I \{p\}\, c\, \{q\}$, if for any state graphs G, G', $G \models_I p$ and $\langle c, G\rangle \rightarrow^* G'$ imply $G' \models_I q$. We call $\{p\}\, c\, \{q\}$ *valid*, denoted as $\models \{p\}\, c\, \{q\}$, if $\models_I \{p\}\, c\, \{q\}$ under any interpretation I.

We say a specification sequent $\Phi \vdash \Psi$ is *true* under an interpretation I, if Φ is true under I implies Ψ is true under I. Naturally, a set of specifications Φ is true under I means each specification of Φ is true under I.

Recall that a hypothesis of a logic rule is either an assertion, a specification or a specification sequent. We establish the soundness of logic rules of $\mathcal{H}$ by the following theorem.

Theorem 1 (Soundness of Logic Rules). *Rules (1) to (16) are sound. Here, a logic rule is* sound *means: for any interpretation I, the hypotheses of the rule are true under I implies the conclusion of the rule is true under I.*

Notice that the soundness of a logic rule with no hypothesis simply means the validity of the specification as the conclusion of the rule. The proof of this theorem can be found in our technical report [19].

As a natural deduction of Theorem 1, the proof system $\mathcal{H}$ is *sound*. That is, every specification proved by $\mathcal{H}$ is valid.

Theorem 2 (Soundness). $\vdash \{p\}\, c\, \{q\}$ *implies* $\models \{p\}\, c\, \{q\}$.

6 Conclusions

We propose a graph-based Hoare Logic for reasoning about OO programs. Specifically, the Hoare proof system consists of a set of logic rules that covers most OO constructs such as object creation, local variable declaration and recursive method invocation. We have proved the soundness of the logic that every specification proved by the system is valid.

A distinct feature of the logic is its underlying graph-based operational semantics where execution states of OO programs are visualized as directed and labeled graphs [9]. The simplicity and intuitiveness of graphs improve people's understanding of OO concepts and are thus helpful in thinking of and formulating clear assertions. On the other hand, the graph model is expressive enough to characterize important OO properties such as aliasing and reachability.

As for graph models of OO programs, there is some work that proposes an OO execution semantics [8,3]. However, a graph in their model is a mixture of class structure, object configuration together with commands to be executed and thus difficult to comprehend. It is not clear either how assertions can be formulated and reasoned about. The notion of trace in our graph model comes from [7], a trace model for pointers and objects. But the main concern of their work is to maintain the aliasing information.

There is some work on Hoare logic for reasoning about OO programs. In particular, Pierik and De Boer's logic [14] based on term substitution is close to our work. But different from our approach, they calculate the weakest precondition for both assignment and object creation, and a complicated form of substitution for dynamic allocation of objects is needed. Von Oheimb and Nipkow [17] present a machine-checked Hoare logic for a Java-like language in Isabelle. They use a semantic representation of assertions to manipulate the program state explicitly instead of syntactic term substitution. Similarly, Poetzsch-Heffter and Müller [15] use an explicit object store in their logic and present axioms for manipulating the store. Recently, separation logic [16] has been applied for reasoning about OO languages [13]. By heap separation, aliasing can be handled in a natural way and modularity of reasoning is achieved. But meanwhile, users should be careful of information on separation of heaps for writing correct assertions.

Future work includes the proof of the completeness of the logic, i.e., every valid specification can be proved by the system. In fact, we are quite confident that it is complete, because the logic rules are indeed provided to deal with every kind of program constructs and the only difficult case is recursive method invocation. Besides the development of the theory, it is also important to apply the logic to a more substantial case study and further to investigate tool support for application of automated techniques of verification and analysis of OO programs.

References

1. Apt, K., de Bakker, J.: Semantics and proof theory of pascal procedures. In: Salomaa, A., Steinby, M. (eds.) ICALP 1977. LNCS, vol. 52, pp. 30–44. Springer, Heidelberg (1977)
2. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Science of Computer Programming 74(4), 168–196 (2009)
3. Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 383–398. Springer, Heidelberg (2004)
4. He, J., Liu, Z., Li, X.: rCOS: A refinement calculus of object systems. Theor. Comput. Sci. 365(1-2), 109–142 (2006)
5. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
6. Hoare, C.A.R.: Procedures and parameters: An axiomatic approach. In: Symposium on Semantics of Algorithmic Languages. Lecture Notes in Mathematics, vol. 188, pp. 102–116. Springer (1971)
7. Hoare, C.A.R., He, J.: A trace model for pointers and objects. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 1–17. Springer, Heidelberg (1999)
8. Kastenberg, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)
9. Ke, W., Liu, Z., Wang, S., Zhao, L.: A graph-based operational semantics of OO programs. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 347–366. Springer, Heidelberg (2009)
10. Ke, W., Liu, Z., Wang, S., Zhao, L.: Graph-based type system, operational semantics and implementation of an object-oriented programming language. Technical Report 410, UNU-IIST, P.O. Box 3058, Macau (2009), `http://www.iist.unu.edu/www/docs/techreports/reports/report410.pdf`
11. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
12. Liu, Z., Morisset, C., Stolz, V.: rCOS: theory and tools for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 62–80. Springer, Heidelberg (2010)
13. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL 2005, pp. 247–258. ACM, New York (2005)
14. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 64–78. Springer, Heidelberg (2003)

15. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential Java. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 162–176. Springer, Heidelberg (1999)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE Computer Society (2002)
17. von Oheimb, D., Nipkow, T.: Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 89–105. Springer, Heidelberg (2002)
18. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects of Computing 21(1-2), 103–131 (2009)
19. Zhao, L., Wang, S., Liu, Z.: Graph-based object-oriented Hoare logic. Technical Report 458, UNU-IIST, P.O. Box 3058, Macau (2012), http://iist.unu.edu/sites/iist.unu.edu/files/biblio/report458.pdf