WP Semantics for OO Programs and Its Applications[☆]

Liu Yingjing and Qiu Zongyan

LMAM and Department of Informatics, School of Mathematical Sciences, Peking University

Abstract

For the object oriented (OO) world, developing formal semantics for theoretical study and practical use is still an important topic despite of a decade's efforts. In this paper, for a sufficiently large subset of sequential Java with a pure reference semantics model, we define a Weakest Precondition (WP) semantics, and prove its soundness and completeness. Based on this WP semantics, we study specifications of methods and the refinement relationship between specifications, and we propose new definitions for object invariants and behavioral subtyping notation for general OO programs.

Keywords: Object Orientation, Weakest Precondition, Separation Logic, Specification, Refinement, Behavioral Subtyping

1. Introduction

Object Orientation (OO) is widely used in the software development practice. Due to the even higher demands on the reliability and correctness of software systems (or in general, computer-based systems) in recent years, the powerful and useful frameworks for specifying and verifying OO programs are more demanded. For such a framework to be useful, two mutually dependent issues must be considered: a formal semantics for OO programs as the basis for verification, which should be powerful enough to capture desired behaviors of a wide range of typical programs; and a bundle of useful specification and verification techniques, which can support modular verification, thus offering scalability to the OO program verification.

For the semantics, we believe that Weakest Precondition (WP) is among the best choices. Edsger W. Dijkstra introduced WP semantics for procedural programs in his seminal paper [13]. A WP semantics can archive completeness, and define precisely the behavior of the programs. As the result, a WP semantics can be used not only directly to verify programs, but also as a solid foundation for defining important program concepts formally, or to validate verification frameworks and tools. In addition, defining the semantics as the predicate transformers makes it completely independent of implementations. Due to these reason, WP semantics is widely recognized as a solid foundation for formal studies on programs and systems in the procedural world. Many works about semantics, modeling, as well as programming and software development have

[☆]Supported by NNSF of China Grant No. 90718002. This work is a substantial extension of the publication [31] in 8th International Colloquium on Theoretical Aspects of Computing, 2011.

been proposed based on the ideas and techniques of WP, e.g. [32, 1]. After the blooming of OO in the years of 1980s, it is natural to think about the WP semantics for OO programs, and hope that it can play the similar roles here as in the procedural world.

According to our knowledge, the striving for a WP semantics for OO programs began in later 1990. As examples, [12] proposed one of the earliest WP calculus for OO programs, and Cavalcanti and Naumann [10] gave also a WP semantics for an OO language. With more study on OO, various forms of WP semantics for OO programs are proposed and used in various frameworks targeting to specify and verify OO programs. Noticeable works falling into this category include, for example, ESC/JAVA [14], LOOP [8], JML [20] and Spec# [4], which all use WP techniques to generate verification conditions from programs. However, most of these works take the value model for variables, thus do not support object reference and sharing. Many central issues in OO languages and programs, especially those related to mutable object structures, have not been sufficiently addressed. After the emergence of Separation Logic [41], things begin to change. People used WP ideas or techniques in their work, explicitly or implicitly, to deal with various problems for OO programs, e.g, [42, 36]. However, there is still not enough work directly on a WP semantics to establish it as both covering most of the important features of OO programs, and having the well-founded theoretical foundation.

For the second issue, it is well known that *Behavioral Subtyping*, or *Liskov Substitutability Principle*, [25] plays a central role in the formal verification for OO programs. Almost all works in this field adopt this principle as a part of their essential ingredient. Liskov [26] gave a formal treatment for behavioral subtyping by a group of constraint rules, and considered object invariants (or class invariants) in the rules. Some researchers offered various new definitions afterward, where the most influential one is given in the work [22], where Leavens and Naumann proposed a natural refinement order on the specifications, and defined the behavioral subtyping based on that order. More importantly, they proved that behavioral subtyping is equivalent to modular reasoning for OO programs. The object invariants are also considered in their subsequent work [23]. However, we will point out in this paper that their treatment for object invariant need not be improved from the practical view points.

In this paper we will present a deep investigation on a WP semantics for OO programs, and study its applications in various aspects related to the verification of OO programs. We use a language named μJava [40] in the work, which is a sufficiently large subset of sequential Java covering most important OO features, and develop a WP semantics for μJava based on an OO Separation Logic (OOSL) [29]. We present and prove some important properties of this semantics for the WP semantics, especially, we have proved that this semantics to be both sound and complete with respect to an operational semantics. However, because the proof is rather long, we leave it in our report [30]. After the semantics, we use it as a theoretical tool to formalize and study some essential concepts in specification, verification and formal development of OO programs, and then use it to prove the soundness of a set of Hoare-style inference rules. to show its power and usefulness.

The main contributions of this work are as follows:

- Firstly, we have answered a theoretical question: Can we have a sound and complete WP semantics for a typical class-based OO language that takes the pure reference semantic model for variables and fields? The work presented here tells us, the answer is yes.

- We present and prove some important properties of the WP semantics, especially the frame-property on the line of Separation Logic which is very important in supporting local reasoning for the mutable object structures.

- We illustrate such a WP semantics is useful in the OO field by use it to define the specifications and their refinement relationship, to prove some refinement judgments. In addition, we give a new definition for object invariant by this WP semantics, and discuss why this definition is different from the existing ones and is more reasonable and useful.
- We give a new definition for the behavioral subtyping concept, which follows Liskov Substitutability Principle and Leavens's natural refinement order. However, our formalism is more natural and more practically useful than what defined in previous works.
- As an application of the WP semantics, we give a set of Hoare-style inference rules and prove their soundness using the WP semantics. In some related work, we showed that the rules a useful in proving correctness of OO programs.

The rest of the paper is organized as follows. We introduce briefly μJava in Section 2. The WP semantics is defined in Section 3, and some properties are provided and proved, especially the frame property. We present also the soundness and completeness result for the WP semantics in this section. In Section 4, we study behavioral subtyping and other important issues in OO verification, including specification, refinement, object invariant, and give their definitions using the WP semantics. In Section 5 a set of Hoare-style inference rules for μJava is given and proved sound. Then we discuss some related work and conclude the paper. To prevent the paper becomes to long, we left some proofs, especially for the soundness and completeness of the WP semantics, and some more applications of the semantics in our report [30].

2. μJava

For carrying on the study, we use a simple OO language μJava [40] in this work. μJava can be seen as a sequential subset of Java. It takes the pure reference semantics for variables and fields, and covers many important OO features related to the object sharing, updating, creation, etc. thus reflects the essences of mainstream OO languages. The definition of μJava takes a clear separation of store and heap operations. The language is simple for facilitating the theoretical study, and large enough for covering important OO features, e.g., dynamic binding, object sharing, aliasing, casting, etc. The syntax of the language is as follows:

$$\begin{array}{lll}
v & ::= & \texttt{this} \mid x \qquad\qquad\qquad\qquad e \ ::= \ \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid v \\
b & ::= & \texttt{true} \mid \texttt{false} \mid e = e \mid \neg b \mid b \wedge b \mid b \vee b \\
c & ::= & \texttt{skip} \mid x := e \mid x := v.a \mid x := (C)v \mid v.a := e \mid x := v.m(\overline{e}) \mid \\
 & & x := \texttt{new}\ C(\overline{e}) \mid \texttt{return}\ \ e \mid c;c \mid \texttt{if}\ \ b\ c\ \texttt{else}\ \ c \mid \texttt{while}\ \ b\ c \\
T & ::= & \texttt{Bool} \mid \texttt{Object} \mid C \qquad\qquad M \ ::= \ T\ m(\overline{T\,z})\{\overline{T\,y};\ c\} \\
K & ::= & \texttt{class}\ C : C\{\overline{T\ a}; C(\overline{T\,z})\{\overline{T\,y};\ c\}; \overline{M}\} \\
G & ::= & K \mid K\ G
\end{array}$$

Here x denotes a variable, C a class name, a and m field and method names respectively. We use over-lined form to represent a sequence. Here:

- We assume a built-in type `Object`, which has no field, as the supertype of all classes, and `Null` the subtype of any class. `Null` is the type for `null` which used only in the type and semantics definitions. We assume only one primitive type `Bool` = {`true`, `false`} to simplify the formal study, which is not a supertype or subtype of any type.

- μJava offers only restricted expressions those values depend only on the store, and some special forms of assignments, including plain $x := e$, mutation $v.a := e$, and lookup $x := v.a$. We take *cast* as a part of a special form of assignments, as well as $x :=$ `new` $C(\overline{e})$ which creates a new object, builds it with parameters $\overline{e}$ and assigns its reference to variable x. We assume in commands all references to fields of current object in methods are decorated with `this`, to make the field references uniformly of the form $v.a$.

- The special $C(\overline{T\,z})\{\overline{T\,y};\ c\}$ in each class C is the constructor, which has the same name as the class. We assume `return` e only appears as the last statement in non-constructor methods. For recording the return value in semantic definitions, we assume an internal-variable res, which cannot be used in programs. We require that local variables and res initialized to special nil values (represented as nil) according to their types, i.e., rfalse for `Bool` and rnull for class types.

- We do not have access control here. A program is just a sequence of class declarations. There might be a main method in last class as the execution entry. If there is a main method in a program, we say that it is a *closed program*, otherwise it is an *open program*.

For simplify the formal study, we have some simplifications and restrictions on the language. However, most of the general forms of expressions and commands encountered in practical OO languages can be encoded by structures here with the help of auxiliary variables.

In paper [10] a static environment was defined, and then in the definition of the WP semantics, only well-typed expressions and commands were considered. We follow this idea, and define a static environment $\Gamma = (\Delta, \Theta)$ to support (and simplify) the WP definitions. Here component Δ records all static structural information for typing, and Θ records the method declarations in each class for method body lookup in the definition. Both Δ and Θ can be established by scanning the program text. In report [40] we give rules for the construction of Δ and Θ, and rules for typing μJava programs. Because the technique is standard, we omit the details here, but assume only some notations relative to them with respect to the requirements of this paper.

Typing environment Δ_G for program G (abbr. Δ) records static structural information of G. We use $\mathsf{super}(C_1, C_2)$ to mean that C_2 is the immediate superclass of C_1, and $T_1 <: T_2$ as the transitive closure of super, here we omit the context Δ when it is clear. On the other hand, for method body lookup, we will use notation $\Theta, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\}$ to denote that $m(\overline{z})\{\mathsf{var}\ \overline{y}; c\}$ is a method declared in class C with parameters $\overline{z}$, local variables $\overline{y}$ and body c.

We will use $\mathsf{dtype}(v)$ or $\mathsf{dtype}(v.a)$ to denote the declaration type of variable v or field $v.a$. We use $\mathsf{fields}(T)$ to denote the set of field names of type T, and use $\mathsf{fdtypes}(T)$ to denote the map from the field names of T to their corresponding type names.

Type judgments for expressions takes the form $\Gamma, C, m \vdash e : T$ to mean that e is of the type T in the scope of method m of C under Γ; and $\Gamma, C, m \vdash c : \mathbf{com}$ means that c is a well-typed command in the body of m in class C. For method m in C, we use $\Gamma, C \vdash m : \mathbf{method}$ to state that it is well-typed. In the following, we consider only the well-formed commands and methods.

3. A WP Semantics for μJava

In this section, we define a Weakest Precondition (WP) semantics for μJava and investigate its properties. Before the definition, we introduce our assertion language OOSL in the first.

3.1. OOSL: An Object-Oriented Separation Logic

Separation Logic [41] is an extension of Hoare logic, aims to reason programs in C like languages which manipulating mutable data structures. It and its variants have archive great success in various fields. In our previous work [29], we proposed a variant of Separation Logic, OO Separation Logic (OOSL), to describe OO programs' states. We will use it as the assertion language in this work. Here we give a short introduction to OOSL. And readers can refer to the appendix or [29] to find more details for the logic.

We use a revised Stack-Heap storage model to represent the run-time states of OO programs. A state $s = (\sigma, O) \in \text{State}$ consists of a store and a heap (object pool):

$$\text{Store} \;\hat{=}\; \mathsf{Name} \rightharpoonup_{\mathsf{fin}} \mathsf{Ref} \qquad \text{Heap} \;\hat{=}\; \mathsf{Ref} \rightharpoonup_{\mathsf{fin}} \mathsf{Name} \rightharpoonup_{\mathsf{fin}} \mathsf{Ref} \qquad \text{State} \;\hat{=}\; \text{Store} \times \text{Heap}$$

Here Name is an infinite set of names, where special names $\mathtt{true}, \mathtt{false}, \mathtt{null} \in \mathsf{Name}$ denote boolean constants and null respectively. Type is an infinite set of types. $\mathtt{Object}, \mathtt{Null}, \mathtt{Bool} \in \mathsf{Type}$ take the same meaning as in μJava. Ref is an infinite set of references as object identities. It contains three constants: $\mathsf{rtrue}, \mathsf{rfalse}$ refer to the two $\mathtt{Bool}$ objects respectively, and rnull refers to nothing. For any $\sigma \in \text{Store}$, we assume $\sigma\mathtt{true} = \mathsf{rtrue}$, $\sigma\mathtt{false} = \mathsf{rfalse}$ and $\sigma\mathtt{null} = \mathsf{rnull}$. We will use $r, r_1, \ldots$ to denote references, and $a, a_1, \ldots$ for fields of objects.

The assertion language of OOSL is similar to that of Separation Logic, with some revisions to fit the needs of OO programs:

$$\begin{array}{lll}
\rho & ::= & \mathtt{true} \mid \mathtt{false} \mid r_1 = r_2 \mid r : T \mid r <: T \mid v = r \\
\eta & ::= & \mathbf{emp} \mid r_1.a \mapsto r_2 \mid \mathsf{obj}(r, T) \\
\psi & ::= & \rho \mid \eta \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi \Rightarrow \psi \mid \psi * \psi \mid \psi \mathrel{-\!\!*} \psi \mid \exists r \cdot \psi \mid \forall r \cdot \psi
\end{array}$$

where T is a type, v a variable or constant, r_1, r_2 references. We use Ψ to denote the set of OOSL assertions. Here are some explanations:

- ρ denotes assertions independent of heaps. References are atomic values here. For any two references r_1, r_2, $r_1 = r_2$ holds iff r_1 and r_2 are identical. $r : T$ indicates that r refers to an object with exact type T. $r <: T$ means that r refers to an object of T or subtype of T. And $v = r$ asserts that the value of variable or constant v is r.

- η denotes assertions involving heaps: **emp** asserts the heap is empty; and the singleton assertion takes the form $r_1.a \mapsto r_2$, as a cell in heap is a field-value binding of an object (denoted by a reference). In addition, $\mathsf{obj}(r, T)$ indicates that the heap contains exact an entire object of type T, which r refers to. In Separation Logic, people use $l \mapsto -$ or $l \hookrightarrow -$ to denote that location l is allocated in current storage. Because the existence of empty objects in OO, we cannot use $r.a \mapsto -$ or $r.a \hookrightarrow -$ to serve the similar purpose for an object. To solve this problem, we introduce an assertion form $\mathsf{obj}(r, T)$ here.

- Connectors $*$ and $\mathrel{-\!\!*}$ are from Separation Logic: $\psi_1 * \psi_2$ means current heap can be split into two parts, where ψ_1 and ψ_2 hold on each part respectively; $\psi_1 \mathrel{-\!\!*} \psi_2$ means that if we add a heap satisfying ψ_1 to current heap, the combined heap will satisfy ψ_2.

In addition, we allow user-defined predicates to extend vocabulary of the assertion language. A predicate definition takes the form $p(\overline{x}) \doteq \psi$, where p is a symbol (predicate name), $\overline{r}$ are the formal parameters of the predicate, and ψ is the body, which is an assertion correlated with $\overline{r}$.

Recursive definitions are allowed. In fact, these predicates are indispensable to support specification and verification programs involving recursive data structures, e.g., lists, trees, etc. Having a definition for p, $p(\overline{e})$ can be used as a basic assertion.

We use $\psi[v/x]$ (or $\psi[r/x]$, $\psi[r_1/r_2]$) to denote substitution of variable or constant v (or reference r_2). We treat $r = v$ the same as $v = r$, and define $v.a \mapsto r$ as $\exists r' \cdot (v = r' \wedge r'.a \mapsto r)$. Some abbreviations are borrowed from Separation Logic:

$$r.a \mapsto - \;=\; \exists r' \cdot r.a \mapsto r' \qquad r.a \hookrightarrow r' \;=\; r.a \mapsto r' * \mathtt{true}$$

We also need to consider type of the object which r refers in a state, which is denoted by $\mathsf{otype}(r)$.

We use $(\sigma, O) \models \psi$ to mean that assertion ψ holds on state (σ, O), its definition is given in **Appendix A**. When $(\sigma, O) \models \psi$, we say also that state (σ, O) satisfies assertion ψ. We have proved that most axioms and inference rules for Separation Logic are also correct in OOSL.

We give some lemmas below that are useful in our study on the WP semantics. For the proofs, please notice: (1), $(\sigma, O) \models \psi_1 * \psi_2$ iff there exists O_1 and O_2 such that $O = O_1 * O_2$, and $(\sigma, O_1) \models \psi_1$ and $(\sigma, O_2) \models \psi_2$. Note that $O = O_1 * O_2$ implies $O_1 \perp O_2$. (2), $(\sigma, O) \models \psi_1 \mathrel{-\!\!*} \psi_2$ iff for any ψ' such that $\psi' \perp \psi$ and $(\sigma, \psi') \models \psi_1$, then we have $(\sigma, \psi' * \psi) \models \psi_2$.

Lemma 1. $(r.a \mapsto r_1 * (r.a \mapsto r_2 \mathrel{-\!\!*} \psi_1) * \psi_2) \Rightarrow (r.a \mapsto r_1 * (r.a \mapsto r_2 \mathrel{-\!\!*} (\psi_1 * \psi_2)))$

Proof. Suppose $(\sigma, O) \models r.a \mapsto r_1 * (r.a \mapsto r_2 \mathrel{-\!\!*} \psi_1) * \psi_2$, then there exists $O_1 * O_2 * O_3 = O$ such that $(\sigma, O_1) \models r.a \mapsto r_1$, $(\sigma, O_2) \models r.a \mapsto r_2 \mathrel{-\!\!*} \psi_1$, and $(\sigma, O_3) \models \psi_2$. Then, for any O_4 such that $O_4 \perp O_2$ and $(\sigma, O_4) \models r.a \mapsto r_2$, we have $(\sigma, O_2 * O_4) \models \psi_1$. Because O_1 and O_4 have the same domain $r.a$, thus $O_3 \perp O_4$, so we have $(\sigma, O_2 * O_3 * O_4) \models \psi_1 * \psi_2$, and then

$$(\sigma, O) \models r.a \mapsto r_1 * (r.a \mapsto r_2 \mathrel{-\!\!*} (\psi_1 * \psi_2)). \qquad \square$$

A similar law in Separation Logic takes the form $(x \mapsto e_1 * (x \mapsto e_2 \mathrel{-\!\!*} \psi_1) * \psi_2) \Rightarrow (x \mapsto e_1 * (x \mapsto e_2 \mathrel{-\!\!*} (\psi_1 * \psi_2)))$.

Lemma 2. *The following inference rules from* [41] *are sound in OOSL:*

$$(1)\ \frac{\psi_1 * \psi_2 \Rightarrow \psi_3}{\psi_1 \Rightarrow (\psi_2 \mathrel{-\!\!*} \psi_3)} \qquad (2)\ \frac{\psi_1 \Rightarrow (\psi_2 \mathrel{-\!\!*} \psi_3)}{\psi_1 * \psi_2 \Rightarrow \psi_3}$$

Lemma 3. *We have the following inference rules, where ψ is an arbitrary assertion:*

$$(1)\ \frac{\psi_1 \Rightarrow (\psi_2 \mathrel{-\!\!*} \psi_3)}{(\psi_1 * \psi) \Rightarrow (\psi_2 \mathrel{-\!\!*} (\psi_3 * \psi))} \qquad (2)\ \frac{\psi_1 * \psi_2 \Rightarrow \psi_3}{(\psi_1 * \psi) \Rightarrow (\psi_2 \mathrel{-\!\!*} (\psi_3 * \psi))}$$

Proof. We prove (1) here, while (2) can be obtained by (1) and **Lemma 2** (1).

For any σ, O_0 such that $(\sigma, O_0) \models \psi_1 * \psi$, there exist O_1, O satisfying $O_1 \perp O$, $O_1 * O = O_0$, $(\sigma, O_1) \models \psi_1$ and $(\sigma, O) \models \psi$. Take any O_2 such that $O_2 \perp O$ and $(\sigma, O_2) \models \psi_2$, we notice also $O_2 \perp O_1$. From $(\sigma, O_1) \models \psi_1$ and the premise, we have $(\sigma, O_1) \models \psi_2 \mathrel{-\!\!*} \psi_3$. From the choice of O_2, we have $(\sigma, O_2 * O_1) \models \psi_3$. From this we have $(\sigma, O_2 * O_1 * O) \models \psi_3 * \psi$, and then $(\sigma, O_1 * O) \models \psi_2 \mathrel{-\!\!*} (\psi_3 * \psi)$, i.e. $(\sigma, O_0) \models \psi_2 \mathrel{-\!\!*} (\psi_3 * \psi)$. Now we can conclude that

$$(\psi_1 * \psi) \Rightarrow (\psi_2 \mathrel{-\!\!*} (\psi_3 * \psi)). \qquad \square$$

We give some more details in Appendix A. The more complete treatments for OOSL can be found in our previous paper [29] and related report [28].

$$\text{[WP-COND]}\ [\![\Gamma, C, m \vdash \texttt{if}\ b\ c_1\ \texttt{else}\ c_2]\!] = \lambda\psi \cdot (b \Rightarrow [\![c_1]\!]\psi) \wedge (\neg b \Rightarrow [\![c_2]\!]\psi)$$

$$\text{[WP-ITER]}\ [\![\Gamma, C, m \vdash \texttt{while}\ b\ c]\!] = \lambda\psi \cdot \mu\phi \cdot (\neg b \Rightarrow \psi) \wedge (b \Rightarrow [\![c]\!]\phi)$$

$$\text{[WP-SEQ]}\ [\![\Gamma, C, m \vdash c_1; c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!] \qquad \text{[WP-SKIP]}\ [\![\Gamma, C, m \vdash \texttt{skip}]\!] = \lambda\psi \cdot \psi$$

$$\text{[WP-ASN]}\ [\![\Gamma, C, m \vdash x := e]\!] = \lambda\psi \cdot \psi[e/x] \qquad \text{[WP-RET]}\ [\![\Gamma, C, m \vdash \texttt{return}\ \ e]\!] = \lambda\psi \cdot \psi[e/\mathsf{res}]$$

$$\text{[WP-LKUP]}\ [\![\Gamma, C, m \vdash x := v.a]\!] = \lambda\psi \cdot \exists r_1, r_2 \cdot (v = r_1 \wedge r_1.a \hookrightarrow r_2 \wedge \psi[r_2/x])$$

$$\text{[WP-MUT]}\ [\![\Gamma, C, m \vdash v.a := e]\!] = \lambda\psi \cdot \exists r_1, r_2 \cdot (v = r_1 \wedge e = r_2 \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 -\!\!* \psi)))$$

$$\text{[WP-CAST]}\ [\![\Gamma, C, m \vdash x := (N)v]\!] = \lambda\psi \cdot \exists r \cdot (r <: N \wedge v = r \wedge \psi[v/x])$$

$$\text{[WP-MTHD]}\ \frac{\begin{array}{c}\Theta, C, m \twoheadrightarrow \lambda(\bar{z})\{\mathsf{var}\ \bar{y}; c\},\\ [\![\Gamma, C, m \vdash c]\!] = f\end{array}}{\begin{array}{l}[\![\Gamma, C \vdash m : \mathbf{method}]\!]\\ = \lambda\ \texttt{this}, \bar{z} \cdot \lambda\psi \cdot f(\psi)[\overline{\mathsf{nil}}/\bar{y}]\end{array}} \qquad \text{[WP-INV]}\ \frac{\begin{array}{c}\Gamma, C, m_0 \vdash v : T, \quad S_1, ..., S_k \text{ are all subtypes of } T,\\ [\![\Gamma, S_i \vdash m : \mathbf{method}]\!] = F_i\ (i = 1, ..., k)\end{array}}{\begin{array}{l}[\![\Gamma, C, m_0 \vdash x := v.m(\bar{e})]\!]\\ = \lambda\psi \cdot \exists r \cdot (v = r \wedge \bigvee_i (r : S_i \wedge F_i(r, \bar{e})(\psi[\mathsf{res}/x])))\end{array}}$$

$$\text{[WP-NEW]}\ \frac{[\![\Gamma, N \vdash N : \mathbf{method}]\!] = F}{[\![\Gamma, C, m \vdash x := \texttt{new}\ N(\bar{e})]\!] = \lambda\psi \cdot \forall r \cdot (\mathsf{raw}(r, N) -\!\!* F(r, \bar{e})(\psi[r/x]))}$$

Figure 1: WP Semantics for μJava

3.2. The WP Semantics

As what common in the procedural world, we define the WP semantics of a command c as a predicate transformer, which maps any given predicate ψ to the weakest precondition of c with respect to ψ. Due to the type system, we define here the semantics only for well-typed commands, that is, for any command c in the discussion, $\Gamma, C, m \vdash c : \mathbf{com}$ is supposed true. The static necessities ensured by typing will not appear in the semantic rules.

Remember Ψ denotes the set of assertions in OOSL, thus the set of predicate transformers is $\mathcal{T} = \Psi \to \Psi$. We use $[\![\Gamma, C, m \vdash c : \mathbf{com}]\!]$ to denote the WP semantics of command c, and sometimes $[\![c]\!]$ when Γ, C and m are clear from the context. In most cases, we use λ-notations for the definition. We use $f = g$ in the definition to mean that $\forall\psi \cdot f(\psi) \Leftrightarrow g(\psi)$.

The WP semantics rules for μJava commands and methods are given in Figure 1. The semantics of sequential composition, choice, and iteration are routine, given as three rules [WP-SEQ], [WP-COND], and [WP-ITER]. For rule [WP-ITER], $\mu\phi \cdot f$ denotes the least fix-point of $\lambda\phi \cdot f$. Below we give some explanations to each of the other groups of the rules.

Basic Commands. The semantics of `skip` is the identity transformer. The semantics of the plain assignment $x := e$ is ordinary, due to the restricted expression forms in μJava, and the clear separation of assertion forms for the stores and heaps in OOSL. The semantics of return command, described by rule [WP-RET] is the same as an assignment to special variable res.

If any ψ holds after mutation $v.a := x$, it is necessary that variable v points to an object that has field a. The existence of field a is guaranteed by typing. After the assignment, $v.a$ holds the reference which is the value of x. This is defined by rule [WP-MUT]. The last part of the rule takes the similar form as the corresponding rule in the Separation Logic. On the other hand, as specified by rule [WP-LKUP], the lookup command $x := v.a$ is similar to the plain assignment. The only additional pre-requirement for executing this command is that variable v must point to an object (which contains field a) but not nil at that time.

Type cast is treated by rule [WP-CAST]. Here we ask for that the variable v must refer to an object with type N or N's subtype. Remember that for any type T, `null` $<: T$.

Method and Invocation. Before discussing the WP semantics of method invocations, as well as the `new` commands, we need to have some preparation.

A method can be thought as a parameterized command. Based on this idea, we define semantics of a method as a parameterized predicate transformer with type $\mathcal{PT}_{n+1} \mathrel{\hat{=}} \mathsf{Ref}^{n+1} \to \mathcal{T}$, where n is the number of the parameters of the method, and the extra parameter designates current object of the invocation. For a parameterized predicate $F : \mathcal{PT}_{n+1}$, when we apply it to a set of references $r_0, r_1, \ldots, r_n$, which stand for the objects referred by `this` and all the other arguments, we obtain a predicate transformer $F(r_0, r_1, \ldots, r_n)$. For convenience, we define an abbreviation form that for any expression e,

$$F(r_0, .., e, .., r_n) \mathrel{\hat{=}} \lambda\psi \cdot \exists r \cdot (e = r \wedge F(r_0, .., r, .., r_n)(\psi)).$$

We may also accept more than one expressions in this abbreviation. For example, we can see $F(r, \overline{e})$ in the last two rules in Figure 1.

We use the notation $[\![\Gamma, C \vdash m : \mathbf{method}]\!]$, or shortly $[\![C.m]\!]$, to denote the WP semantics of a method m defined in class C. Here m could be C to denote the constructor of the class. Now we are ready to discuss the relative rules.

Rule [WP-MTHD] gives the semantics of methods and constructors. Here all local variables are replaced with nil values. This means that, on one hand, all local variables are initialized with the nil according to the requirements mentioned in **Section 2**. On the other hand, this also makes all the local variables inaccessible from outside of the method. So, if a given ψ contains names in $\overline{y}$, we should rename such local variables to avoid the name captures.

If all methods are non-recursive, we can get their parameterized predicate transformers directly. Otherwise, by the rules, we can obtain a group of equations about parameterized predicate transformers. [16] tells us there exists a least fix-point solution for such a set of equations, and we define the solution as the WP semantics for these methods respectively. So the WP semantics for methods is well-defined.

Based on the above definition, the semantics for invocations is given by rule [WP-INV] which takes a similar form as the corresponding one in [10]. Here we collect methods (with the same name m) of all the relative subclasses in the program (which are determined statically by the program text), and define the weakest precondition as the disjunction of the predicates produced by these subclass methods. Note that $r : S_i$ ensures $r \neq \mathsf{rnull}$. When reasoning on a real invocation, this disjunction will be resolved by the type of current object and then disappears. In building the precondition, we replace x with res in ψ, because the invocation can be viewed as two "actions": the first one is the execution of the body of $v.m(\overline{e})$ which stores the return value in res at the end, and the second copies the value in res to x.

Clearly, this rule demands that the program been reasoned about is a closed program. In this case, our definition can describe the behavior of a method invocation precisely. The closeness of the program is one crucial condition, because only under this condition, the WP semantics can achieve completeness, which we have proved in our report [30]. On the other hand, for an open program, because we can not know how the possibly subclasses will be defined, we may only try some looser definition which can be sound but not complete.

Object Creation. Informally, object creation can be thought as two "actions" sequentially: the first one extends current heap by creating a new raw object (while all its fields take nil values) and obtains its reference; the second initiates the object's state. That is exactly the case for practical OO languages, and specified by rule [WP-NEW]. The rule states that if we append any new object of class N to current heap, after the execution of the constructor, ψ will hold. In this rule, the assertion $\mathsf{raw}(r, N)$ asserts that r refers to a raw object of N, with the definition as

$$\mathsf{raw}(r, N) \,\hat{=}\, \begin{cases} \mathsf{obj}(r, N), & N \text{ has no field} \\ r : N \wedge (r.a_1 \mapsto \mathsf{nil}) * .. * (r.a_k \mapsto \mathsf{nil}), & \{a_1, .., a_k\} = \mathsf{fields}(N) \end{cases}$$

We will use $\mathsf{raw}(r, -)$ if do not care the type. We can prove this assertion satisfies the following proposition, which says that separated objects must be different:

Proposition 1. $\mathsf{raw}(r_1, \text{-}) * \mathsf{raw}(r_2, \text{-}) \Rightarrow r_1 \neq r_2$. □

3.3. Properties and Examples

Now we prove that the WP semantics for μJava is well defined, i.e., it forms a well-defined function on all well-typed commands. In addition, the predicate transformers defined by all the well-typed commands are monotone functions. Then we give some examples to show how the WP semantic can be used directly to verify programs.

In the first, we have the following theorems.

Theorem 1. *Suppose we have built environment Γ_P for a program P. For any well-typed command c with $\Gamma, C, m \vdash c : \mathbf{com}$, its semantics $[\![\Gamma, C, m \vdash c : \mathbf{com}]\!]$ is a total function on all formulas. Additionally, if $\Gamma, C \vdash m : \mathbf{method}$, the semantics $[\![\Gamma, C \vdash m : \mathbf{method}]\!]$ is a well-defined parameterized predicate transformer.*

Proof. By induction on the structures of the commands. We will show that there is a semantic definition for each typing derivation.

Skip, Assignment, Mutation, Lookup, Cast, and Return: Semantics for these commands are direct, so the conclusion holds.

Sequential Composition, Condition, Iteration, Method: In each case, there is a direct semantic definition and the conclusion holds by induction hypothesis.

Method Invocation: Because the command is well-typed, by the typing rules and hypothesis, we have that for every class S_i <: T, m is a valid method of S_i, so there exists f_i that $F_i = \lambda \mathtt{this}, \overline{z} \cdot \lambda\psi \cdot f_i(\psi)[\overline{\mathsf{nil}}/\overline{y}]$ is the parameterized predicate transformer for $S_i.m$. The conclusion holds.

Object Creation: Similar to Method Invocation, by the typing rule and hypothesis, we have that there exists f that $F = \lambda \mathtt{this}, \overline{z} \cdot \lambda\psi \cdot f(\psi)[\overline{\mathsf{nil}}/\overline{y}]$ is the parameterized predicate transformer for N's constructor. Then the conclusion holds.

Based on above proof, we can get the conclusion for methods immediately. □

Theorem 2. *Suppose $f : \mathcal{T}$ is a predicate transformer produced by WP rules given in* Figure 1*, and ψ, ψ' are any well-formed predicates. If $\psi \Rightarrow \psi'$, then $f(\psi) \Rightarrow f(\psi')$.*

Proof. By induction on the structure of the commands.

Sequential Composition, Condition, Iteration: By induction hypothesis we can get the conclusion for each case.

Skip: The proof is trivial.

Assignment, Cast, and Return: It is trivial that substitution of variables in predicate does not change its value. So the conclusion holds.

Mutation: By the definition of the separation conjunction and separation implication, the conclusion holds.

Lookup: Since $\psi \Rightarrow \psi'$, then $\psi[r_2/x] \Rightarrow \psi'[r_2/x]$, so we have:

$$\begin{array}{l}(\exists r_1, r_2 \cdot (v = r_1) \wedge (r_1.a \hookrightarrow r_2) \wedge \psi[r_2/x]) \\ \Rightarrow (\exists r_1, r_2 \cdot (v = r_1) \wedge (r_1.a \hookrightarrow r_2) \wedge \psi'[r_2/x]).\end{array}$$

Method Invocation: Suppose $[\![\Gamma, T \vdash m : \mathbf{method}]\!] = F = \lambda\mathtt{this}, \bar{z} \cdot \lambda\psi \cdot f(\psi)[\overline{\mathsf{nil}}/\bar{y}]$. Because $\psi \Rightarrow \psi'$, by induction hypothesis, we have $f(\psi) \Rightarrow f(\psi')$. Thus $f(\psi)[\overline{\mathsf{nil}}/\bar{y}] \Rightarrow f(\psi')[\overline{\mathsf{nil}}/\bar{y}]$ holds, then $F(v, \bar{e})(\psi) \Rightarrow F(v, \bar{e})(\psi')$. By properties of $*$ and $\wedge$, the conclusion holds.

Object Creation: Suppose $[\![\Gamma, C \vdash N : \mathbf{method}]\!] = F = \lambda\mathtt{this}, \bar{z} \cdot \lambda\psi \cdot f(\psi)[\overline{\mathsf{nil}}/\bar{y}]$, where $[\![\Gamma, N, N \vdash c : \mathbf{com}]\!] = f$. Because $\psi \Rightarrow \psi'$, by induction hypothesis, we have $f(\psi) \Rightarrow f(\psi')$. Thus $f(\psi)[\overline{\mathsf{nil}}/\bar{y}] \Rightarrow f(\psi')[\overline{\mathsf{nil}}/\bar{y}]$ holds, then $F(v, \bar{e})(\psi) \Rightarrow F(v, \bar{e})(\psi')$. By properties of $-\!\!*$, the conclusion holds. □

3.4. Examples

Now we give some simple examples to illustrate the semantics defined in this section. We do not list all codes (such as constructors) in the example, because they are not necessary in our discussion in the following.

Example 1 (Empty Object Creation). Now we show how to do verification with the WP semantics. Because an instance of `Object` is an empty, we suppose the body of the constructor of `Object` is `skip`, then by the WP semantics:

$$[\![\Gamma, \mathtt{Object} \vdash \mathtt{Object} : \mathbf{method}]\!] = \lambda\psi \cdot \psi$$

Then we have the following calculation:

$$\begin{array}{rl} & [\![x := \mathtt{new}\ \mathtt{Object}(); y := \mathtt{new}\ \mathtt{Object}();]\!](x \neq y) \\ = & [\![x := \mathtt{new}\ \mathtt{Object}();]\!](\forall r \cdot \mathsf{raw}(r, \mathtt{Object}) -\!\!* x \neq r) \\ = & \forall r_1, r_2 \cdot \mathsf{raw}(r_1, \mathtt{Object}) -\!\!* \mathsf{raw}(r_2, \mathtt{Object}) -\!\!* r_1 \neq r_2 \\ = & \mathtt{true} \end{array}$$

This indicates that two newly created empty objects are different. It also shows our accurate treatment for empty objects. □

Example 2 (Iteration). Suppose we have the code for classes *Nd* and *Iter* in Figure 2. By the weakest precondition semantics, we have:

$$\begin{array}{rl} & [\![\mathtt{while}\ (p! = \mathtt{null})\ p := p.a]\!] \\ = & \lambda\psi \cdot \mu\phi \cdot (p = \mathsf{rnull} \Rightarrow \psi) \wedge (p \neq \mathsf{rnull} \Rightarrow [\![p = p.a]\!]\phi) \\ = & \lambda\psi \cdot \mu\phi \cdot (p = \mathsf{rnull} \Rightarrow \psi) \wedge (p \neq \mathsf{rnull} \Rightarrow \\ & \qquad (\exists r_1, r_2 \cdot p = r_1 \wedge r_1.a \hookrightarrow r_2 \wedge \phi[r_2/p])) \end{array}$$

```
Class Nd : Object { Nd a; }
Class Iter : Object {
  Nd h;
  Bool m(){
    Nd p; p := this.h;
    while (p ! = null) { p := p.a; }
    return true;
  }
}
Class Reft : Object {
  Bool m(){
    C1 a1, C2 a2, C3 a3;
    a1 := new C1(); a3 := (C3)a1;
    a2 := (C2)a3; return true;
  }
}
```

```
Class A : Object {
  T a, b;
  T m(){ T t; t := this.a; return t; }
}
Class B : A {
  T m(){ T t; t := this.b; return t; }
}
Class C {
  A r; B s; T x; T1 y, z; Eu, v;
  A n(){ x := s.m(); y.f := x;
      x := r.m(); z.f := x;
      u := new E(); v := new E();
      return r; }
}
```

Figure 2: Some Simple Examples

Let

$$f(\phi) = (p = \mathsf{rnull} \Rightarrow \psi) \wedge (p \neq \mathsf{rnull} \Rightarrow (\exists r_1, r_2 \cdot p = r_1 \wedge r_1.a \hookrightarrow r_2 \wedge \phi[r_2/p]))$$

By Tarski's theorem about fix-point, we have

$$\mu\phi \cdot f(\phi) = \bigvee f^i(\mathtt{false})$$

We expand some $f^i(\mathtt{false})$:

$$\begin{array}{lcl} f(\mathtt{false}) & = & p = \mathsf{rnull} \Rightarrow \psi \\ f^1(\mathtt{false}) & = & (p = \mathsf{rnull} \Rightarrow \psi) \wedge (p \neq \mathsf{rnull} \Rightarrow (\exists r \cdot p = r \wedge r.a \hookrightarrow \mathsf{rnull} \Rightarrow \psi)) \\ \ldots & & \end{array}$$

So, for describing $\mu\phi \cdot f(\phi)$, we define a predicate $\mathsf{access}(r_1, a, r_2)$, which assert that we can reach r_2 going forward along field a from reference r_1:

$$\mathsf{access}(r_1, a, r_2) \mathrel{\hat{=}} r_1 = r_2 \vee (\exists r_3 \cdot r_1.a \hookrightarrow r_3 \wedge \mathsf{access}(r_3, a, r_2)). \tag{1}$$

With this definition, we can obtain that the fix-point

$$\mu\phi \cdot f(\phi) = \exists r \cdot p = r \wedge \mathsf{access}(r, a, \mathsf{rnull}) \wedge \psi[\mathsf{rnull}/p]$$

So we get the semantics of the `while` command in method m is:

$$\begin{array}{cl} & [\![\mathtt{while}\ (p! = \mathtt{null})\ p := p.a]\!] \\ = & \exists r \cdot p = r \wedge \mathsf{access}(r, a, \mathsf{rnull}) \wedge \psi[\mathsf{rnull}/p] \end{array}$$

□

Example 3 (Reference Types). Suppose we have C_1 <: C_2 <: C_3, and $C_1()$ is the constructor of C_1. Consider the code for class *Reft* in Figure 2. One can see that from **Theorem 1**, we have that method m in class *Reft* has a well defined semantics. But in [10], due to the absence of object sharing, the semantics of downcast assignment $a_2 := (C_2)a_3$ is undefined. □

Example 4 (Method Invocation). Suppose we have the declarations of class A, B, C as shown in Figure 2, and n is a method of class C. Additionally, suppose that class $T1$ has an attribute f of type T. Using the definitions and the reasoning rules of OO Separation Logic, we can reach the following results:

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\Gamma, C, n \vdash x := s.m(); y.f := x : \mathbf{com}]\!]} \\
= \lambda\psi\cdot & \exists r \neq \mathsf{rnull} \cdot s = r \wedge r : B \wedge \\
& (\exists r_1, r_2 \cdot s = r_1 \wedge r_1.b \hookrightarrow r_2 \wedge \\
& \ (\exists r_3, r_4 \cdot y = r_3 \wedge r_2 = r_4 \wedge (r_3.f \mapsto - * (r_3.f \mapsto r_4 -\!\!* \psi[r_4/x])))) \\
= \lambda\psi\cdot & \exists r \neq \mathsf{rnull}, r_2, r_3 \cdot s = r \wedge y = r_3 \wedge r : B \wedge \\
& r.b \hookrightarrow r_2 \wedge (r_3.f \mapsto - * (r_3.f \mapsto r_2 -\!\!* \psi[r_2/x])). \\
\multicolumn{2}{l}{[\![\Gamma, C, n \vdash x := r.m(); z.f := x : \mathbf{com}]\!]} \\
= \lambda\psi\cdot & \exists r \neq \mathsf{rnull}, r_2, r_3 \cdot s = r \wedge z = r_3 \wedge \\
& ((r : A \wedge r.a \hookrightarrow r_2 \wedge (r_3.f \mapsto - * (r_3.f \mapsto r_2 -\!\!* \psi[r_2/x]))) \vee \\
& \ (r : B \wedge r.b \hookrightarrow r_2 \wedge (r_3.f \mapsto - * (r_3.f \mapsto r_2 -\!\!* \psi[r_2/x])))).
\end{array}
$$

□

3.5. Frame Property of the WP Semantics

Frame rule [41] is very important in Separation Logic for local reasoning. We have the similar property for the WP semantics.

Theorem 3. *Given command c and assertions ψ_1, ψ_2, if $FV(\psi_2) \cap md(c) = \emptyset$, then*

$$([\![c]\!]\psi_1) * \psi_2 \Rightarrow [\![c]\!](\psi_1 * \psi_2)$$

where $FV(\psi_2)$ is the set of all program variables (including internal variable res*) in ψ_2, $md(c)$ is the variable set modified by c, defined as:*

$$
md(c) = \begin{cases}
\{x\}, & c \text{ is } x := \ldots \\
\{\mathsf{res}\}, & c \text{ is } \texttt{return} \ \ldots \\
md(c_1) \cup md(c_2), & c \text{ is } c_1; c_2 \\
md(c_1) \cup md(c_2), & c \text{ is } \texttt{if}\ b\ c_1\ \texttt{else}\ c_2 \\
md(c), & c \text{ is } \texttt{while}\ b\ c \\
\emptyset, & \text{otherwise}
\end{cases}
$$

Proof. The proof is conducted by induction on the structures of commands. In the deduction, we need to use some equivalence or implication laws of OOSL. In the proof, we will use ψ to denote $[\![c]\!]\psi_1$ in some of the cases.

Skip: The result is trivially true.

Assignment "$x := e$": For the antecedent, we have $[\![x := e]\!]\psi_1 * \psi_2 = \psi_1[e/x] * \psi_2$. Because ψ_2 does not contains x, then $\psi_1[e/x] * \psi_2 \Leftrightarrow (\psi_1 * \psi_2)[e/x] = [\![x := e]\!](\psi_1 * \psi_2)$. The case for the **Return** statement is similar except the assigned variable is res.

Mutation "$v.a := x$": Here $\psi = \exists r_1, r_2 \cdot (v = r_1 \wedge x = r_2 \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 -\!\!* \psi_1)))$. By the properties of $*$, $-\!\!*$ and **Lemma 1**, we have deduction

$$
\begin{array}{l}
\exists r_1, r_2 \cdot (v = r_1 \wedge x = r_2 \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 -\!\!* \psi_1))) * \psi_2 \\
\Rightarrow \exists r_1, r_2 \cdot ((v = r_1 * \psi_2) \wedge (x = r_2 * \psi_2) \wedge ((r_1.a \mapsto - * (r_1.a \mapsto r_2 -\!\!* \psi_1)) * \psi_2)) \\
\Rightarrow \exists r_1, r_2 \cdot (v = r_1 \wedge x = r_2 \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 -\!\!* (\psi_1 * \psi_2)))).
\end{array}
$$

This is exactly $[\![v.a\ :=\ x]\!](\psi_1 * \psi_2)$.

Lookup "$x\ :=\ v.a$**":** Here $\psi = \exists r_1, r_2 \cdot (v = r_1 \wedge r_1.a \hookrightarrow r_2 \wedge \psi_1[r_2/x])$. Because ψ_2 does not contain variable x, and by the properties of operator $*$ and $\wedge$, we can easily obtain:

$$\begin{aligned}
&\exists r_1, r_2 \cdot (v = r_1 \wedge r_1.a \hookrightarrow r_2 \wedge \psi_1[r_2/x]) * \psi_2\\
\Rightarrow\ &\exists r_1, r_2 \cdot ((v = r_1 * \psi_2) \wedge (r_1.a \hookrightarrow r_2 * \psi_2) \wedge (\psi_1[r_2/x] * \psi_2))\\
\Rightarrow\ &\exists r_1, r_2 \cdot (v = r_1 \wedge r_1.a \hookrightarrow r_2 \wedge (\psi_1 * \psi_2)[r_2/x]).
\end{aligned}$$

Cast "$x\ :=\ (N)v$**":** Here $\psi = (\exists r \cdot (r <: N) \wedge \psi_1[r_2/x])$, because ψ_2 does not contain variable x, and with the properties of operator $*$ and $\wedge$, the conclusion holds.

Sequential Composition "$c_1;\ c_2$**":** By induction hypothesis and **Theorem 2**, we have:

$$(\psi * \psi_2) \Rightarrow [\![c_1]\!]([\![c_2]\!]\psi_1 * \psi_2) \Rightarrow [\![c_1]\!]([\![c_2]\!](\psi_1 * \psi_2)) = [\![c_1; c_2]\!](\psi_1 * \psi_2).$$

Condition "`if` $b\, c_1$ `else` c_2**":** Here $\psi = (b \Rightarrow [\![c_1]\!]\psi_1) \wedge (\neg b \Rightarrow [\![c_2]\!]\psi_1)$. By the induction hypotheses, and the relations among $*$, $\wedge$, and $\vee$, we have deduction

$$\begin{aligned}
(\psi * \psi_2)\ &\Rightarrow\ ((b \Rightarrow [\![c_1]\!]\psi_1) * \psi_2) \wedge ((\neg b \Rightarrow [\![c_2]\!]\psi_1) * \psi_2)\\
&\Rightarrow\ ((\neg b * \psi_2) \vee ([\![c_1]\!]\psi_1 * \psi_2)) \wedge ((b * \psi_2) \vee ([\![c_2]\!]\psi_1 * \psi_2))\\
&\Rightarrow\ (b \Rightarrow ([\![c_1]\!]\psi_1 * \psi_2)) \wedge (\neg b \Rightarrow ([\![c_2]\!]\psi_1 * \psi_2))\\
&\Rightarrow\ (b \Rightarrow ([\![c_1]\!](\psi_1 * \psi_2))) \wedge (\neg b \Rightarrow ([\![c_2]\!](\psi_1 * \psi_2)))
\end{aligned}$$

Iteration "`while` $b\ c$**":** Let $\psi' = \mu\phi \cdot (\neg b \Rightarrow \psi_1) \wedge (b \Rightarrow [\![c]\!]\phi)$, then we have $\psi = \psi' = (\neg b \Rightarrow \psi_1) \wedge (b \Rightarrow [\![c]\!]\psi')$. By induction hypothesis we have $[\![c]\!]\psi' * \psi_2 \Rightarrow [\![c]\!](\psi' * \psi_2)$. By the relations among $*$, $\wedge$ and $\vee$, we have:

$$\begin{aligned}
(\psi * \psi_2) \Rightarrow (\psi' * \psi_2)\ &\Rightarrow\ ((\neg b \Rightarrow \psi_1) * \psi_2) \wedge ((b \Rightarrow [\![c]\!]\psi') * \psi_2)\\
&\Rightarrow\ ((b * \psi_2 \vee \psi_1 * \psi_2)) \wedge ((\neg b * \psi_2 \vee [\![c]\!]\psi' * \psi_2))\\
&\Rightarrow\ (\neg b \Rightarrow \psi_1 * \psi_2) \wedge (b \Rightarrow [\![c]\!]\psi' * \psi_2)\\
&\Rightarrow\ (\neg b \Rightarrow \psi_1 * \psi_2) \wedge (b \Rightarrow [\![c]\!](\psi' * \psi_2))
\end{aligned}$$

By Knaster-Tarski's least fix-point theorem, we have

$$(\psi' * \psi_2) \Rightarrow \mu\phi \cdot ((\neg b \Rightarrow \psi_1 * \psi_2) \wedge (b \Rightarrow [\![c]\!]\phi))$$

So the conclusion holds.

Invocation "$x\ :=\ v.m(\overline{e})$**":** Here $\psi\ =\ \exists r \cdot (v = r \wedge \bigvee_i (r : S_i \wedge F_i(r, \overline{e})(\psi_1[\mathsf{res}/x])))$. By the relations among $*$, $\wedge$ and $\vee$, we have

$$\begin{aligned}
&(\exists r \cdot v = r \wedge (\bigvee_i (r : S_i \wedge F_i(r, \overline{e})(\psi_1[\mathsf{res}/x])))) * \psi_2\\
\Rightarrow\ &\exists r \cdot (v = r * \psi_2) \wedge (\bigvee (r : S_i \wedge (F_i(r, \overline{e})(\psi_1[\mathsf{res}/x])) * \psi_2))\\
\Rightarrow\ &\exists r \cdot (v = r) \wedge (\bigvee (r : S_i \wedge (F_i(r, \overline{e})(\psi_1[\mathsf{res}/x]) * \psi_2)))
\end{aligned}$$

Notice here ψ_2 does not contain res, and by the induction hypothesis, $F_i(r, \overline{e})(\psi_1[\mathsf{res}/x]) * \psi_2 \Rightarrow F_i(r, \overline{e})((\psi_1 * \psi_2)[\mathsf{res}/x])$, then we have the conclusion.

Object Creation "x := new $N(\overline{e})$": Here $\psi * \psi_2 = (\forall r \cdot \mathsf{raw}(r, N) -\!\!* F(r, \overline{e})(\psi_1[r/x])) * \psi_2$. Suppose $\psi * \psi_2$ holds, we need to prove that $(\forall r \cdot \mathsf{raw}(r, N) -\!\!*(F(r, \overline{e})((\psi_1 * \psi_2)[r/x])))$ holds too. By the induction hypothesis and the properties of $*$ and $-\!\!*$, we have deduction:

$$
\begin{array}{rl}
& ((\forall r \cdot \mathsf{raw}(r, N) -\!\!*(F(r, \overline{e})(\psi_1[r/x]))) * \psi_2) * \mathsf{raw}(r', N) \\
\Rightarrow & ((\forall r \cdot \mathsf{raw}(r, N) -\!\!*(F(r, \overline{e})(\psi_1[r/x]))) * \mathsf{raw}(r', N)) * \psi_2 \\
\Rightarrow & (F(r, \overline{e})(\psi_1[r/x])) * \psi_2 \\
\Rightarrow & F(r, \overline{e})((\psi_1 * \psi_2)[r/x])
\end{array}
$$

By **Lemma 3**, the conclusion holds. **NOT CLEAR ENOUGH! NEED more DETAILS!**

Now, we finish the proof. □

This theorem is used in proving the frame rules of inference systems below. It is very important in studying the local reasoning for OO programs.

3.6. Soundness and Completeness

In this subsection we give the soundness and completeness theorems of the WP semantics defined above. We leave their proofs in our report [30].

We take **COM** the space of legal commands, and use $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$ to denote configuration transformation of μJava, that says when command c executes from current state (σ, O), after its execution of the state will be (σ', O'). A formal definition for the state transformation (i.e., an operational semantics) is given also in our report [30].

In the following definitions, we see the WP as a generator which produces for each command in **COM** a predicate transformer. Informally, a WP semantics is *sound* if the following statement holds for any pair of a well-typed command c and a predicate ψ, if c executes from a state satisfying the weakest precondition ψ' of c with respect to ψ, and if c terminates, the final state will satisfy ψ. It is formally defined as follows.

Definition 1 (Soundness). A given WP predicate transformer $[\![\bullet]\!] : \mathcal{T}$ is *sound*, if and only if for any $\psi, \psi' \in \Psi$ and $c \in$ **COM** satisfying $[\![\Gamma, C, m \vdash c : \mathbf{com}]\!]\psi = \psi'$, we have: For any pair of states (σ, O) and (σ', O'), if $(\sigma, O) \models \psi'$ and $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, then $(\sigma', O') \models \psi$.

Theorem 4 (Soundness Theorem)**.** *The WP semantics for* μJava *given in* Section 3.2 *is sound.*

Informally, a WP semantics is *complete*, if it really gives the weakest precondition. Equivalently, for any well-typed command c, if from any state s, we can reach a state satisfying the postcondition ψ by executing c, then $\psi' = [\![c]\!]\psi$ holds on s. It is formally defined as follows.

Definition 2 (Completeness). A given WP predicate transformer $[\![\bullet]\!] : \mathcal{T}$ is *complete*, if and only if for any $\psi, \psi' \in \Psi$ and $c \in$ **COM** satisfying $[\![\Gamma, C \vdash c : \mathbf{com}]\!]\psi = \psi'$, we have: For any pair of states (σ, O) and (σ', O'), if $(\sigma', O') \models \psi$ and $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, then $(\sigma, O) \models \psi'$.

Theorem 5 (Completeness Theorem)**.** *The WP semantics for* μJava *given in* Section 3.2 *is complete.*

Having the sound and complete WP semantics for μJava defined, in the following, we will use it as a tool to study various theoretical problems related to OO programs.

4. Specification and Behavioral Subtyping

Because the WP semantics defined in is both sound and complete, we can use it as a foundation to study various theoretical problems. In this section we study some important problems related to OO program verification: code and method specification, refinement, object invariants, and the behavioral subtyping concept.

4.1. Specification and Refinement

In literatures on program verification, people often use a pair of assertions, namely P and Q, as the specification of a programs segment, e.g., a command, a piece of code, or a method. In the following we will use $\{P\}\text{-}\{Q\}$ to denote a specification where P and Q, as usually, are called the precondition and postcondition, respectively.

A command c satisfies the specification $\{P\}\text{-}\{Q\}$, if c executes from a pre-state (σ, O) where $(\sigma, O) \models P$, if c terminates and the resulting state is (σ', O'), then $(\sigma', O') \models Q$. These can be defined based on the WP semantics:

Definition 3 (Command Satisfies Specification). For a command c with $[\![c]\!] = f$, and a pair of predicates P and Q, if $P \Rightarrow f(Q)$, we say that c is partial correct with respect to precondition P and postcondition Q, written $\{P\}\,c\,\{Q\}$.

Please note here we have an implicit static environment and an execution context. We will use this simplified notations in the discussion for conciseness.

Method $C.m(\overline{z})$ satisfies $\{P(\texttt{this}, \overline{z})\}\text{-}\{Q(\texttt{this}, \overline{z})\}$, if $C.m$ executes from a pre-state (σ, O) where $(\sigma, O) \models P(x, \overline{r})$ where x points to the calling object and $\overline{r}$ are the real arguments, if $C.m$ terminates on (σ', O'), then $(\sigma', O') \models Q(x, \overline{r})$. For the constructor, it is similar.

Definition 4 (Method/Constructor Satisfies Specification). For method $m(\overline{z})$ of a class C and a pair of parameterized predicates $P(\texttt{this}, \overline{z})$ and $Q(\texttt{this}, \overline{z})$, assume $[\![C.m(\overline{r'})]\!] = F(r_0, \overline{r'})$. If $\forall r_0, \overline{r'} \cdot P(r_0, \overline{r'}) \Rightarrow F(r_0, \overline{r'})(Q(r_0, \overline{r'}))$, then we say that m is partial correct with respect to the pre/postconditions $P(\texttt{this}, \overline{z})$ and $Q(\texttt{this}, \overline{z})$, written as $\{P(\texttt{this}, \overline{z})\}\,C.m(\overline{z})\,\{Q(\texttt{this}, \overline{z})\}$.

For a class C and two parameterized predicates $P(\overline{z})$ and $Q(\texttt{this}, \overline{z})$, assume $[\![C.C(\overline{r'})]\!] = F(\overline{r'})$. If $\forall r_0, \overline{r} \cdot P(\overline{r}) \Rightarrow (\mathsf{raw}(r_0, C) \mathrel{-\!\!*} F(\overline{r})(Q(r_0, \overline{r})))$, then we say that the constructor of C is partial correct with respect to the pre/postconditions $P(\texttt{this}, \overline{z})$ and $Q(\texttt{this}, \overline{z})$, written as $\{P(\overline{z})\}\,C(\overline{z})\,\{Q(\texttt{this}, \overline{z})\}$.

In the following, we may omit the parameters. Based on the WP semantics and above definitions, now we give a formal definition for the specification refinements.

Definition 5 (Refinement of Specifications). We say a specification $\{P_2\}\text{-}\{Q_2\}$ refines another specification $\{P_1\}\text{-}\{Q_1\}$, written $\{P_1\}\text{-}\{Q_1\} \sqsubseteq \{P_2\}\text{-}\{Q_2\}$, iff for any command c (or method), $\{P_2\}\,c\,\{Q_2\}$ implies $\{P_1\}\,c\,\{Q_1\}$.

This definition implies that, if $\{P_2\}\text{-}\{Q_2\}$ refines $\{P_1\}\text{-}\{Q_1\}$, we can use former specification to substitute the latter anywhere in programs. This definition for the refinement follows the natural refinement order given by Leavens [22].

Proposition 2 (Hoare Refinement). *If $P_1 \Rightarrow P_2$ and $Q_2 \Rightarrow Q_1$, then $\{P_1\}\text{-}\{Q_1\} \sqsubseteq \{P_2\}\text{-}\{Q_2\}$.*

However, although the **Definition 5** for refinement relation is simple and clear, it is not easy for us to use in practice, because we can hardly have ways to investigate all commands. On the other hand, condition $\Gamma \vdash P_C \Rightarrow P_B \wedge Q_B \Rightarrow Q_C$ does not take the heap extention into account. Here we provide a stronger sound condition for refinement judgement.

Theorem 6. *Given specification $\{P_1\}\text{-}\{Q_1\}$ and $\{P_2\}\text{-}\{Q_2\}$, we have $\{P_1\}\text{-}\{Q_1\} \sqsubseteq \{P_2\}\text{-}\{Q_2\}$ if there exists an assertion R such that: (1). R does not contains program variables, and (2). $(P_1 \Rightarrow P_2 * R) \wedge (Q_2 * R \Rightarrow Q_1)$.*

Proof. For any command c, suppose $\{P_2\}\, c\, \{Q_2\}$, then by **Definition 3** we have $P_2 \Rightarrow [\![c]\!](Q_2)$. Because R do not contains program variables, by **Theorem 3**, we have

$$P_2 * R \Rightarrow [\![c]\!](Q_2) * R \Rightarrow [\![c]\!](Q_2 * R).$$

By $(P_1 \Rightarrow P_2 * R) \wedge (Q_2 * R \Rightarrow Q_1)$ and **Theorem 2**, we have $P_1 \Rightarrow [\![c]\!](Q_2 * R) \Rightarrow [\![c]\!](Q_1)$. Then by **Definition 3** and **5**, we have the conclusion. □

In fact, this theorem provides a useful way to check refinement relation in OO programs. Please note that this theorem gives also only a sufficient, but not necessary condition for checking the refinement relation. It is a combination of **Proposition 2** (of Hoare Logic) and the Frame Rule of Separation Logic, and takes the heap and heap extension into account. The condition asks that the storage mentioned in the refining specification can be lesser than what mentioned in the refined specification. This is reasonable.

4.2. Object Invariant

Object invariant is an important concept in both practice and research, because it describes a set of consistent object states we can rely on. An object invariant is popularly thought as a part of the postcondition of constructor and default pre/post conditions for every public methods of the class for simplifying the specifications. This leads to the following verification conditions for class C with object invariant I:

(1) the postcondition of C's constructor implies I;

(2) for every public method $C.m$ with specification $\{P\}\text{-}\{Q\}$, $\{P \wedge I\}\, C.m\, \{Q \wedge I\}$ holds.

(2) means that the "real" specification of a method with written specification $\{P\}\text{-}\{Q\}$ is $\{P \wedge I\}\text{-}\{Q \wedge I\}$. But we want to say that this treatment is not adequate for practice.

We use the code in Figure. 3 to illustrate our points, where three classes are declared: *Base* has a field a and methods f, g, h. Its object invariant demand that a always holds `true`. *Derive* inherits *Base* with a new field b, and strengthens the object invariant with that b should also hold `true`. *Client* may use *Base* and *Derive*. Here the `rep` modifier denotes a *representation field* of a class, that is inaccessible to the clients of the class. This notation comes from the study for *ownership types* [34], and is adopted by many work on object invariants.

Firstly, a method may be invoked by methods of same class, including itself (recursive call). In these circumstances, the object invariant can be ignored, that is, we may allow breaking the invariant for a while within a method. For example, before invoking *Base.f* in *Base.g*, we do assignment `this.`a = `false`, this breaks the invariant temporarily. Under the common idea for invariants, invocation `this.`$f()$ is invalid, although *Base.g* re-establishes the invariant before its termination, although this temporary broken is harmless. This scenario shows that a method

```
class Base : Object {
  rep Bool a;
  invariant this.a ↪ true;
  Base() { this.a = true; }
  void f()
    requires this.a ↪ -;
    ensures this.a ↪ true;
    { this.a = true ; }
  void g()
    requires this.a ↪ -;
    ensures true;
    { this.a = false; this.f(); }
  void h(C c)
    requires this.a ↪ -;
    ensures true;
    { this.a = false; c.fun(this);
      this.a = true; }
}
class Derive : Base {
  rep Bool b;
  invariant this.a ↪ true∗
      this.b ↪ true;
  Derive()
    { this.a = true; this.b = true; }
}
class Client : Object {
  void m(Base b)
    requires b.a ↪ -;
    ensures true
    { b.f(); }
}
```

Figure 3: Sample Code with Specifications and Object Invariants

invocation may appear in two kinds of circumstances, either inside or outside the class where it is declared. The safety requirements for these two circumstances are different. As far as our knowledge, this problem is not well studied.

Secondly, the object invariant is often strengthened in the subclasses. If we treated the precondition of a method as the conjunction of the object invariant and the declared precondition in the method interface, then the preconditions of subclass's methods would become stronger. In the example of Figure 3, if we took ($\mathtt{this}.a \hookrightarrow \mathtt{-} \land \mathtt{this}.a \hookrightarrow \mathtt{true}$) as the precondition for *Base*.f and ($\mathtt{this}.a \hookrightarrow \mathtt{-} \land \mathtt{this}.a \hookrightarrow \mathtt{true} \land \mathtt{this}.b \hookrightarrow \mathtt{true}$) for *Derive*.$f$, we would find that *Derive*.f is not a refinement of *Base*.f! Because for the refinement relation, always we demand that precondition may only be weakened in a subclass. The proposed solutions to this problem are abstracting away the details of the objects [2, 17, 33]. For example, Barnett *et al* [2] suggest to introduce a model field *st* in the specifications to indicate whether the object invariant holds. For this example, as his suggestion, we need to write ($\mathtt{this}.a \hookrightarrow \mathtt{-} \land \mathtt{this}.st = \mathit{Valid}$) as the precondition of *Base*.f, thus it can be validly inherited by *Derive*. Although this works, it still taking the (abstract) object invariant into the method specifications. In addition, to introduce model fields into the work, many model assignments should be carefully arranged.

At last, the object invariant should be transparent for clients. In fact, clients often care only about the specification declared in a method's interface, but have no obligation to establish the object invariant before some invocation. Consider method *Client.m* in above example, the invocation $b.f()$ is valid according *Base*.f's specification. But if we took the idea that "the real specification of *Base*.f is $\mathtt{this}.a \hookrightarrow \mathtt{-} \land \mathtt{this}.a \hookrightarrow \mathtt{true}$", this invocation would become invalid. Barnett *et al* [2] also considered this problem, but because their methodology need also to combine objects invariants with specifications, the validity of $b.f()$ cannot be proved.

From the above examples and relative analysis, we conclude that it is not adequate to treat an object invariants as a necessary part of method specifications. In fact, an object invariant has the class scope, while a specification is only for a particular method, they are independent with each other, and thus should be verified separately. Section 4 defines the verification conditions for a method specification. Here we give a definition for the object invariant:

Definition 6 (Object Invariant). Assertion I is an object invariant of class C, written $C \models I$, iff:

(1) $\forall r, \overline{r'} \cdot ([\![C.C]\!](r, \overline{r'})\mathtt{true}) \Rightarrow ([\![C.C]\!](r, \overline{r'})(I[r/\mathtt{this}]))$; and

(2) for every client accessible method $C.m$, we have $\forall r, \overline{r'} \cdot (([\![C.m]\!](r, \overline{r'})\mathtt{true}) \wedge I[r/\mathtt{this}]) \Rightarrow ([\![C.m]\!](r, \overline{r'})I[r/\mathtt{this}])$.

We will write $C \models I$ to state that I is an object invariant of C.

Here (1) requires that constructor establish the object invariant, and the second condition ensures that every client accessible method preserves the invariant. Thus, the object invariant will always hold for clients. Please note that, this definition for the invariant does not involve method specifications, in addition, for a method, we need only verify that it satisfy *its specification*, and then condition (2) additionally if it is client accessible.

With this definition, the object invariant becomes a self-contained concept. Any assertion satisfying these two conditions is an object invariant of the class. The definition does not mention method specifications at all. Comparing to existing treatments for object invariants, such as [23, 2] and so on, our definition is clearly more adequate. It captures the nature of object invariants, in a sense that it is a complete definition for the concept of object invariant.

Back to our sample code, by definitions of method specification and object invariant, we can conclude the code meets its specifications except of method *Base.h*. We point that, although *Base.h* preserve the object invariant, its implementation is invalid because the object invariant does not hold before calling *c.fun*(`this`). Barnett *et al* studied this problem in [2], and proposed a pair of new primitives "pack/unpack" to explicitly establish or break the object invariant. By the proposed techniques, they can prove that the method call is invalid. We will not discuss this issue further, because it is out of this paper's scope.

4.3. Behavioral Subtyping

The concept of *behavior subtyping* was introduced by Liskov in [25]. The concept is widely recognized extremely important for the correct OO programming practice, as well as a key concept in the formal verification of OO programs. Now we give our definition for *behavioral subtyping* in our framework based on above discussion.

Definition 7 (Behavioral Subtype). Given class C and B, we say C is a behavioral subtype of B, written $C \preceq B$, iff, (1), for every assertion I, $B \models I$ implies $C \models I$; and (2), for every client accessible method $B.m$ we have for any specification $\{P\}\text{-}\{Q\}$, $\{P\}\,B.m\,\{Q\}$ implies $\{P\}\,C.m\,\{Q\}$.

The first condition requires that every object invariant of superclass is an object invariant of subclass; and the second requires that subclass obeys superclass's behavior. Clearly, this definition follows the thought of Liskov substitution principle.

Theorem 7. *If $D \preceq C$ and $C \preceq B$, then $D \preceq B$.*

Now we focus on practical verification procedures. We will develop the verification conditions for a program with respect to the behavioral subtyping requirement.

At first, we introduce some notations. For a μJava program G with method specifications and object invariants, we can build an environment Π_G to record all method specifications declared in the classes under consideration, and an invariant environment Λ_G to record object invariants declared for the classes. We will omit subscript G from now on. More precisely, Π is a map from method/constructor names to their specifications. We think the specifications in Π is indexed by $C.m$ or C, respectively, and use $\{P\}\,C.m\,\{Q\} \in \Pi$ (or $\{P\}\,C.C\,\{Q\} \in \Pi$ for constructor) to state that $\{P\}\text{-}\{Q\}$ is the specification for method $C.m$ (or the constructor of C). On the other hand, Λ maps each class to its object invariant, and $\Lambda(C)$ gives the object invariant declared for class C.

Definition 8 (Satisfaction of specification and invariant environment). We say a program G satisfies specification environment Π and invariant environment Λ, written $G \models (\Pi, \Lambda)$, iff, (1), Π contains one specification for each method in G, and Λ contains one assertion for each class in G; (2) for every $\{P\}\,C.m\,\{Q\} \in \Pi$, $\{P\}\,C.m\,\{Q\}$ holds, here m could be the constructor; and (3), for every class C in G, $C \models \Lambda(C)$ holds.

This definition leads the following verification conditions for $G \models (\Pi, \Lambda)$:

Theorem 8. *Given a program G, a specification environment* Π *and an invariant environment* Λ *satisfying condition* (1) *of* **Definition 8***, we have* $G \models (\Pi, \Lambda)$*, if* (1)*, for every constructor specification* $\{P\}\,C.C\,\{Q\} \in \Pi$*,* $\{P\}\,C.C\,\{Q \wedge \Lambda(C)\}$ *holds, and* (2)*, for every method specification* $\{P\}\,C.m\,\{Q\} \in \Pi$*,* $\{P\}\,C.m\,\{Q\}$ *holds; and if C.m is client accessible,* $\{P \wedge \Lambda(C)\}\,C.m\,\{\Lambda(C)\}$ *holds.*

From this theorem, we can see that suppose $\Lambda(C) = I$, for the constructor of C with specification $\{P\}\text{-}\{Q\}$, we should verify that $\{P\}\,C.C\,\{Q \wedge I\}$; and for a method $C.m$ with specification $\{P\}\text{-}\{Q\}$, we have two proof obligations: $\{P\}\,C.m\,\{Q\}$ for its behavior, and $\{P \wedge I\}\,C.m\,\{I\}$ for the object invariant. This is very different from common treatment like [23, 2] in two aspects: First, method invocations now only rely on declared specifications $\{P\}\text{-}\{Q\}$, but not $\{P \wedge I\}\text{-}\{Q \wedge I\}$ with object invariant considered. Second, the object invariant is hidden inside a class and transparent to clients, but at any program point clients can assert the object invariant hold.

Although **Definition 7** gives a sound definition for behavioral subtype, it is not practical for real verification. As seen, we provide specifications and object invariants for classes with Π and Λ, and usually use them to do verification so that we need not explore the class details. In this sense, Π and Λ give the strongest specifications and object invariants. So we have the following behavioral subtype definition for practice:

Definition 9. Given specification environment Π and invariant environment Λ for program G, and two classes C and B in G, we say, in program G, C is a behavioral subtype of B under Π and Λ, written $(\Pi, \Lambda) \vdash C \leq B$, iff, (1), $\Lambda(C) \Rightarrow \Lambda(B)$, and (2), for every client accessible method $B.m$ we have $\Pi(B.m) \sqsubseteq \Pi(C.m)$.

In fact, this definition implies **Definition 7** providing that Π *and* Λ *give the strongest specifications and object invariants* for all methods and classes in G. And clearly, this behavioral subtyping relationship is transitive.

Definition 10. Given a program G with specification environment Π and invariant environment Λ, we say G satisfies the behavioral subtyping requirement under Π and Λ, iff for any pair of classes $C <: B$, we have $(\Pi, \Lambda) \vdash C \leq B$.

By **Definition 9** and its transitiveness, we can deduce that we only need check the behavioral subtype relationship for immediate super/sub classes. And, combining **Definition 10** and **Theorem 8**, we can obtain a kind of verification conditions for programs with behavioral subtyping requirements. These verification conditions have similar forms as Liskov's [26] and Leavens's [23], but the meanings are very different according to above discussions.

$$\underline{\text{[H-SEQ]}}\ \frac{\Gamma \vdash \{P\}\, c_1\, \{Q\}, \quad \Gamma \vdash \{Q\}\, c_2\, \{R\}}{\Gamma \vdash \{P\}\, c_1; c_2\, \{R\}} \qquad \underline{\text{[H-COND]}}\ \frac{\Gamma \vdash \{b \wedge P\}\, c_1\, \{Q\}, \quad \Gamma \vdash \{\neg b \wedge P\}\, c_2\, \{Q\}}{\Gamma \vdash \{P\}\, \mathtt{if}\ b\, c_1\ \mathtt{else}\ c_2\, \{Q\}} \qquad \underline{\text{[H-ITER]}}\ \frac{\Gamma \vdash \{b \wedge I\}\, c\, \{I\}}{\Gamma \vdash \{I\}\, \mathtt{while}\, b\, c\, \{\neg b \wedge I\}}$$

$$\underline{\text{[H-SKIP]}}\ \Gamma \vdash \{P\}\, \mathtt{skip}\, \{P\} \qquad \underline{\text{[H-ASN]}}\ \Gamma \vdash \{P[e/x]\}\, x := e\, \{P\}$$

$$\underline{\text{[H-MUT]}}\ \Gamma \vdash \{v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto -\}\, v.a := e\, \{v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto r_2\}$$

$$\underline{\text{[H-LKUP]}}\ \Gamma \vdash \{v = r_1 \wedge r_1.a \mapsto r_2\}\, x := v.a\, \{x = r_2 \wedge v = r_1 \wedge r_1.a \mapsto r_2\}$$

$$\underline{\text{[H-CAST]}}\ \Gamma \vdash \{v = r \wedge r <: N\}\, x := (N)v\, \{x = r\} \qquad \underline{\text{[H-RET]}}\ \Gamma \vdash \{P[e/\mathsf{res}]\}\, \mathtt{return}\ e\, \{P\}$$

$$\underline{\text{[H-INV]}}\ \frac{\Gamma \vdash v <: C \qquad \{P\}\, C.m(\overline{z})\, \{Q\} \in \Pi}{\Gamma \vdash \{v = r \wedge \overline{e = r'} \wedge P[r, \overline{r'}/\mathtt{this}, \overline{z}])\}\, x := v.m(\overline{e})\, \{Q[r, \overline{r'}, x/\mathtt{this}, \overline{z}, \mathsf{res}]\}}$$

$$\underline{\text{[H-NEW]}}\ \frac{\{P\}\, C(\overline{z})\, \{Q\} \in \Pi}{\Gamma \vdash \{\overline{e = r'} \wedge P[\overline{r'}/\overline{z}]\}\, x := \mathtt{new}\, C(\overline{e})\, \{\exists r \cdot x = r \wedge Q[r, \overline{r'}/\mathtt{this}, \overline{z}]\}}$$

Figure 4: Inference Rules for Basic Commands and Compositions

5. Hoare Rules for μJava

In this section, as another application of the WP semantics, we define a set of Hoare style inference rules for μJava, and prove their soundness using the WP semantics. We extend the μJava language with specification features in the first:

$$\begin{array}{lcl} S & ::= & \mathtt{requires}\ P; \mathtt{ensures}\ Q \\ M & ::= & T\ m(\overline{T\, z})\, S\, \{\overline{T\, y};\ c;\} \\ K & ::= & \mathtt{class}\ C : C\{\overline{T\, a};\ C(\overline{T\, z})\, S\, \{\overline{T\, y}; c\}; \overline{M}\} \end{array}$$

Here category S denotes the specifications for methods and constructors, where the OOSL assertions P and Q are the pre and post conditions respectively. Both method and constructor declarations are extended with specification structures. Based on this extension, we can investigate the specification and verification problems in the OO field.

For reasoning OO programs with specifications, we must record method specifications. As in the last section, we extend the statical environment Γ with a specification environment Π to record all method specifications of the classes, which takes the form as:

$$\begin{array}{lcl} \pi & ::= & \{P(\mathtt{this}, \overline{z})\}\, C.m(\overline{z})\, \{Q(\mathtt{this}, \overline{z})\} \ \mid\ \{P(\overline{z})\}\, C(\overline{z})\, \{Q(\mathtt{this}, \overline{z})\} \\ \Pi & ::= & \epsilon \mid \Pi, \pi \end{array}$$

Here ϵ denotes the empty environment, m is a method of class C. The pre and post conditions of methods are recorded as parameterized assertions. Here $\{P(\overline{z})\}\, C(\overline{z})\, \{Q(\mathtt{this}, \overline{z})\}$ denotes the specification for the constructor of class C, where the precondition does not mention `this`. We may omit arguments `this` and $\overline{z}$ of parameterized predicates in discussion if they are not important. Clearly, Π can be built statically by simply scanning the program text. We will use notation $\{P\}\, C.m\, \{Q\} \in \Pi$ (or $\{P\}\, C\, \{Q\} \in \Pi$, for constructor) as in the last section.

5.1. Inference Rules

A statement in our inference systems takes the form of $\Gamma \vdash \{P\}\, c\, \{Q\}$ (or $\Gamma \vdash \{P\}\, C.m\, \{Q\}$) to state that $\{P\}\, c\, \{Q\}$ (or $\{P\}\, C.m\, \{Q\}$ where m is a method in class C) is formally provable under Π by the system. When $\Gamma \vdash \{P\}\, c\, \{Q\}$ (or $\Gamma \vdash \{P\}\, C.m\, \{Q\}$) can be derived using the inference system, we say that the triple $\{P\}\, c\, \{Q\}$ (or $\{P\}\, C.m\, \{Q\}$) is Hoare provable.

[H-CONSTR]
$$\frac{\begin{array}{c}\Gamma, C, C \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \Gamma \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \mathsf{raw}(\mathtt{this}, C) * P[\overline{r}/\overline{z}]\}\, c\, \{Q[\overline{r}/\overline{z}]\}\end{array}}{\Gamma \vdash \{P\}\, C(\overline{z})\, \{Q\}}$$

[H-INTRO]
$$\frac{\begin{array}{c}m \text{ is introduced in } C, \quad \Gamma, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \Gamma \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge P[\overline{r}/\overline{z}]\}\, c\, \{Q[\overline{r}/\overline{z}]\}\end{array}}{\Gamma \vdash \{P\}\, C.m(\overline{z})\, \{Q\}}$$

[H-INH]
$$\frac{\begin{array}{c}C \text{ inherits } m \\ \Gamma \vdash \mathsf{super}(C, B) \\ \Gamma \vdash \{P\}\, B.m(\overline{z})\, \{Q\}\end{array}}{\Gamma \vdash \{P\}\, C.m(\overline{z})\, \{Q\}}$$

[H-OVR]
$$\frac{\begin{array}{c}m \text{ is overridden in } C, \quad \Gamma, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\}, \quad \Gamma \vdash \mathsf{super}(C, B) \\ \{P_C\}\, C.m(\overline{z})\, \{Q_C\} \in \Pi, \quad \{P_B\}\, B.m(\overline{z})\, \{Q_B\} \in \Pi \\ \Gamma \vdash \{P_B\}\text{-}\{Q_B\} \sqsubseteq \{P_C\}\text{-}\{Q_C\}, \quad \Gamma \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge P_C[\overline{r}/\overline{z}\}\, c\, \{Q_C[\overline{r}/\overline{z}]\}\end{array}}{\Gamma \vdash \{P_C\}\, C.m(\overline{z})\, \{Q_C\}}$$

Side condition for the rules: local variables $\overline{y}$ are not free in P,Q

Figure 5: Rules for Method and Constructor Declarations

[H-THIS] $\Gamma, C, m \vdash \mathtt{this} : C$

[H-OLD]
$$\frac{\{P\}\, C.m(\overline{z})\, \{Q\} \in \Pi, \quad \Gamma, C, m \vdash \overline{z = r} \wedge P[\overline{r}/\overline{z}] \Rightarrow e = r'}{\Gamma, C, m \vdash \mathbf{old}(e) = r'}$$

[H-CONSQ]
$$\frac{P \Rightarrow P', \quad \Pi \vdash \{P'\}\, c\, \{Q'\}, \quad Q' \Rightarrow Q}{\Gamma \vdash \{P\}\, c\, \{Q\}}$$

[H-EX]
$$\frac{\Gamma \vdash \{P\}\, c\, \{Q\}, \quad r \text{ is free in } P, Q}{\Gamma \vdash \{\exists r \cdot P\}\, c\, \{\exists r \cdot Q\}}$$

[H-FRAME]
$$\frac{\{P\}\, c\, \{Q\}, \quad FV(R) \cap MD(c) = \emptyset}{\{P * R\}\, c\, \{Q * R\}}$$

Figure 6: Additional Inference Rules

Figure 4 lists the rules for basic commands and composition structures. The first three rules for compositions are exact the same as the rules in Hoare logic. Rules [H-SKIP] and [H-ASN] are simple. Rules [H-LKUP] and [H-MUT] have the similar forms as their counterparts in Separation Logic. Rule [H-CAST] is special in the OO world for the type casting, and rule [H-RET] for the return statement which states that it behaves exactly as an assignment to res.

The last two rules in Figure 4 are for the method invocation and object creation, respectively. Notice that we demand that the specification of the overriding method in a subclass satisfies some additional relation with the counterpart of the same method in the direct superclass (see rules in Figure 5), so in rule [H-INV], we requires only the verification by the declare type of variable v. On the other hand, rule [H-NEW] is clear.

The second group contains rules for verifying method and constructor declarations, which are listed in Figure 5. These rules tell us how to check if a specification in Π is consistent with its code, thus is valid in Π. Here is a side condition that in using the rules, we require that the local variables $\overline{y}$ do not occur free in the specification. The condition can be resolved by suitable renaming local variables to some fresh names. The recursive method definitions is allowed here, because body command c may contain invocations to the same method, or invocations which will lead to mutual recursions. In Rule [H-CONSTR], $\mathtt{this} = r \wedge \mathsf{raw}(r, C)$ specifies that `this` refers to a newly created raw object. Then, the command c has its effects on the state of the new object. We have three different rules for the introduction, overriding and inheritance of methods in a class, respectively. Rule [H-INTRO] is simple, which asks for the verification of the body command c in a suitable context. Rule [H-INH] is trivial. For the overriding, rule [H-OVR] asks for additional the verification of $\Gamma \vdash \{P_B\}\text{-}\{Q_B\} \sqsubseteq \{P_C\}\text{-}\{Q_C\}$.

We have some additional inference rules, as shown in Figure 6. Rules [H-THIS] and [H-OLD] are given for typing `this` and evaluation expression e is the pre-state. Rules [H-CONSQ] and [H-EX] are the same as in Hoare Logic. At last, we have the Frame Rule which has the exact the same form as in Separation Logic, where $FV(R)$ is the set of all variables (with res taken into account) in assertion R, and $md(c)$ denotes the variables modified by c.

Before considering the soundness of these rules, we show their usage by an example.

Example 5. Now, we consider the **Example ??** again using our rules. By an assumption that `Object` has an empty constructor, we have first that

$$\{\mathbf{emp}\}\,\mathtt{Object}\,\{\exists r \cdot \mathtt{this} = r \wedge \mathsf{raw}(r, \mathtt{Object})\}$$

is a valid specification for the constructor of `Object`. By [H-NEW] and [H-FRAME], as well as rules for **emp** and $*$, we have the following deduction:

$$\begin{array}{l}
\{\mathtt{true}\}\\
\{\mathbf{emp} * \mathtt{true}\}\\
\ x\ :=\ \mathtt{new\,Object}();\\
\{\exists r \cdot x = r \wedge \mathsf{raw}(r, \mathtt{Object}) * \mathtt{true}\}\\
\ y\ :=\ \mathtt{new\,Object}();\\
\{\exists r_1, r_2 \cdot x = r_1 \wedge y = r_2 \wedge \mathsf{raw}(r_1, \mathtt{Object}) * \mathsf{raw}(r_2, \mathtt{Object}) * \mathtt{true}\}\\
\{x \neq y\}
\end{array}$$

5.2. Soundness of Hoare Triples

Because the WP semantics defined in Section 3.2 is both sound and complete, we define and prove the soundness of our inference rules based on the WP semantics. In Section 4.3, we define the concepts that a command satisfies a specification consisting of a precondition P and a postcondition Q, as a logic statement $P \Rightarrow [\![c]\!]Q$. We have similar definition for the methods and constructors. Although we also use Hoare-triples in that section, to avoid confusion, in this section we use the triple notation, i.e., $\{P\}\,c\,\{Q\}$, only for the Hoare provable triples.

Definition 11 (Soundness of Hoare Rules). For any given set of Hoare style inference rules, we say it is sound, if and only if, for any program G with environment Γ:

- For any command c in G, if $\Gamma \vdash \{P\}\,c\,\{Q\}$, then $\Gamma \vdash P \Rightarrow [\![c]\!]Q$;
- For any method $C.m$ in G, if $\Gamma \vdash \{P\}\,C.m(\overline{z})\,\{Q\}$, then $\Gamma \vdash \forall r_0, \overline{r} \cdot (P[r_0, \overline{r}/\mathtt{this}, \overline{z}] \Rightarrow [\![C.m]\!](\overline{r})(Q[r_0, \overline{r}/\mathtt{this}, \overline{z}]))$.
- For the constructor of any C in G, if $\Gamma \vdash \{P\}\,C(\overline{z})\,\{Q\}$, then $\Gamma \vdash \forall r_0, \overline{r} \cdot (P[\overline{r}/\overline{z}] \Rightarrow (\mathsf{raw}(\mathtt{this}, N) \mathrel{-\!\!*} [\![C.C]\!](\overline{r})(Q[r_0, \overline{r}/\mathtt{this}, \overline{z}])))$.

Here we write Γ explicitly for the OOSL assertions, i.e., in $\Gamma \vdash P \Rightarrow [\![c]\!]Q$. Recall that $[\![C.m]\!]$ is a parameterized predicate transformer, thus the notation $[\![C.m]\!](\overline{r})$ denotes a predicate transformer. The situation is similar for the constructors.

We now prove that the rules given in **Section 5.1** is sound by our WP semantics.

Theorem 9. *The inference rules given in* Figure 4, 5, *and* 6 *are sound.*

Proof. We prove the soundness result by induction, because recursive defined methods are allowed here. We first prove inference rules for commands, and then the others.

[H-SEQ]: Suppose for $\Gamma \vdash \{P\}\,c_1\,\{Q\}$ and $\Gamma \vdash \{Q\}\,c_2\,\{R\}$, we have $P \Rightarrow [\![c_1]\!]Q$ and $Q \Rightarrow [\![c_2]\!]R$ respectively, then $P \Rightarrow [\![c_1]\!][\![c_2]\!]R = [\![c_1; c_2]\!]R$.

[H-COND], [H-ITER], [H-SKIP], [H-ASN], [H-RET], [H-EX], [H-CONSQ]: The proofs are similar to what for Hoare rules in procedural language, and monotonicity of WP semantics. We omit them here.

[H-MUT]: By the WP semantics

$$\begin{aligned}
&\llbracket v.a \ := \ e\rrbracket(v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto r_2)\\
=\ &\exists r_3, r_4 \cdot v = r_3 \wedge e = r_4 \wedge (r_3.a \mapsto - * (r_3.a \mapsto r_4 \mathrel{-\!\!*} (v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto r_2)))\\
=\ &v = r_1 \wedge e = r_2 \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 \mathrel{-\!\!*} r_1.a \mapsto r_2)\\
=\ &v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto -.
\end{aligned}$$

[H-LKUP]: By the WP semantics

$$\begin{aligned}
&\llbracket x \ := \ v.a\rrbracket(x = r_2 \wedge v = r_1 \wedge r_1.a \mapsto r_2)\\
=\ &\exists r_3, r_4 \cdot v = r_3 \wedge r_3.a \hookrightarrow r_4 \wedge (x = r_2 \wedge v = r_1 \wedge r_1.a \mapsto r_2) \ = \ v = r_1 \wedge r_1.a \mapsto r_2
\end{aligned}$$

[H-CAST]: By the WP semantics, $\llbracket x \ := \ (N)v\rrbracket(x = r) = \exists r \cdot r <: N \wedge v = r$, this can be deduced by $v = r \wedge r <: N$, so the conclusion holds.

[H-INV]: By WP semantics, we have

$$\begin{aligned}
&\llbracket x := v.m(\overline{e})\rrbracket(Q(r, \overline{r'})[x/\mathsf{res}])\\
=\ &\exists r \cdot (v = r) \wedge (\textstyle\bigvee(r : S_i \wedge \llbracket S_i.m\rrbracket(r, \overline{e})((Q(r, \overline{r'})[x/\mathsf{res}])[\mathsf{res}/x])))\\
=\ &\exists r \cdot (v = r) \wedge (\textstyle\bigvee(r : S_i \wedge \llbracket S_i.m\rrbracket(r, \overline{e})(Q(r, \overline{r'}))))
\end{aligned}$$

By the hypothesis, and the specifications are well-specified, so for all $i = 1...k$, we have $P(\mathtt{this}, \overline{z}) \Rightarrow (\llbracket S_i.m\rrbracket(\mathtt{this}, \overline{z}))(Q(\mathtt{this}, \overline{z}))$. Then

$$\begin{aligned}
v = r \wedge \overline{e = r'} \wedge P(r, \overline{r'}) \ &\Rightarrow \ v = r \wedge \overline{e = r'} \wedge (\textstyle\bigvee r : S_i) \wedge (\textstyle\bigwedge(\llbracket S_i.m\rrbracket Q(r, \overline{r'})))\\
&\Rightarrow \ v = r \wedge \overline{e = r'} \wedge (\textstyle\bigvee(r : S_i \wedge \llbracket S_i.m\rrbracket Q(r, \overline{r'})))\\
&= \ \llbracket x := v.m(\overline{e})\rrbracket(Q(r, \overline{r'})[x/\mathsf{res}]).
\end{aligned}$$

[H-NEW]: By the WP semantics we have

$$\begin{aligned}
&\llbracket x := \mathtt{new}\ N(\overline{e})\rrbracket(\exists r \cdot x = r \wedge Q(r, \overline{r'}))\\
=\ &\forall r \cdot \mathsf{raw}(r, N) \mathrel{-\!\!*} (\llbracket N\rrbracket(r, \overline{e}))((\exists r' \cdot x = r' \wedge Q(r, \overline{r'}))[r/x])\\
=\ &\forall r \cdot \mathsf{raw}(r, N) \mathrel{-\!\!*} (\llbracket N\rrbracket(r, \overline{e}))Q(r, \overline{r'})\\
=\ &\forall r \cdot \mathsf{raw}(r, N) \mathrel{-\!\!*} (\llbracket c\rrbracket Q(r, \overline{r'})[\overline{\mathsf{nil}}/\overline{y}]
\end{aligned}$$

By the hypothesis and rule [H-CONSTR], we have

$$(\overline{y = \mathsf{nil}} \wedge \mathtt{this} = r \wedge \mathsf{raw}(r, N) * P(\overline{z})) \Rightarrow \llbracket c\rrbracket(Q(r, \overline{z}))$$

By the definition of $\mathrel{-\!\!*}$, we have

$$\begin{aligned}
(\mathtt{this} = r \wedge P(\overline{z})) \ &\Rightarrow \ (\mathsf{raw}(r, N) \mathrel{-\!\!*} \llbracket c\rrbracket(Q(r, \overline{z})))[\overline{\mathsf{nil}}/\overline{y}]\\
&\Rightarrow \ (\mathsf{raw}(r, N) \mathrel{-\!\!*} (\llbracket c\rrbracket(Q(r, \overline{z}))[\overline{\mathsf{nil}}/\overline{y}])
\end{aligned}$$

At last, because r is an arbitrary reference, we have the conclusion.

[H-CONSTR]: By the premise, induction hypothesis, and above proofs, we have $\mathsf{raw}(r, C) * P[\overline{r}/\overline{z}] \Rightarrow \llbracket c\rrbracket(Q[\overline{r}/\overline{z}])$. By rule [WP-MTHD], we have

$$\mathsf{raw}(r, C) * P[\overline{r}/\overline{z}] \Rightarrow (\llbracket C.C\rrbracket(\mathtt{this}, \overline{r}))(Q[\overline{r}/\overline{z}]).$$

By **Lemma 2**, and because r is an arbitrary reference, we can obtain that

$$\forall \overline{r} \cdot (P[\overline{r}/\overline{z}] \Rightarrow (\mathsf{raw}(r, C) \mathrel{-\!\!*} (\llbracket C.C\rrbracket(\mathtt{this}, \overline{r}))(Q[\overline{r}/\overline{z}]))).$$

[H-INTRO]: Here m is introduced into the inheritance hierarchy of C, thus we can consider it locally. Similar to the proof for [H-CONSTR], by the premise we have

$$\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge P[\overline{r}/\overline{z}] \Rightarrow [\![c]\!](Q[\overline{r}/\overline{z}]).$$

This is $P[\overline{r}/\overline{z}] \Rightarrow ([\![c]\!](Q(\overline{r}/\overline{z}))[\overline{\mathsf{nil}}/\overline{y}])$. Now by the rule [WP-MTHD], we have $P[\overline{r}/\overline{z}] \Rightarrow ([\![c]\!]Q(\mathtt{this}, \overline{z}))[\overline{\mathsf{nil}}/\overline{y}]$. Then by the arbitrary of $\overline{r}$ we have the conclusion.

[H-OVR]: The proof is similar to [H-INTRO], where the additional premise $\Gamma \vdash \{P_B\}\text{-}\{Q_B\} \sqsubseteq \{P_C\}\text{-}\{Q_C\}$ ensures that $C.m$ satisfies $\{P\}\text{-}\{Q\}$, thus ensures the soundness of [H-INV].

[H-INH]: An inherited method can be viewed as an overriding method which has exact the same specification and implementation as what in the superclass. We can prove [H-INH] following the line for [H-OVR] and this idea.

[H-FRAME]: The conclusion comes directly from **Theorem 3**.

Now we can conclude that the inference rules given in Section 5.1 are sound. □

6. Related Work

In this paper, we investigate various important concepts and techniques in OO program verification: WP semantics, specification refinement, object invariant and behavioral subtyping, inference rules for OO programs, etc. Here we discuss some related work.

6.1. WP Semantics

WP semantics [13] is one of the most powerful tools in theoretical study on procedural programs and related research areas. Researchers apply WP technique to reasoning programs, define and validate semantics and formal frameworks, generate verification conditions, validate refinement rules, and so on. However, after many years of striving in the OO world, WP has not yet showed its potentials as in the procedural world. After all, a satisfactory and well-studied WP semantics does still not emerge yet. This is the motivation of the work presented here.

To our limited knowledge, efforts towards a WP semantics for OO programs since 1999, while in [12] a WP calculus was given for OO programs aiming to support some object sharing. The definition of the semantics is restricted to the forms of syntactic substitution. For example, the semantics for assignments is defined as:

$$l.x[e/x] \mathrel{\hat{=}} \mathsf{if}\ (l[e/x]) = \mathsf{self}\ \mathsf{then}\ e\ \mathsf{else}\ (l[e/x]).x\ \mathsf{fi}$$

where $l = \mathsf{self}$ stands for that l refers to current object. The authors made many restrictions to the programs and assertions, and paid high prices on various special cases in the work. However, because even $l = \mathsf{self}$ cannot be checked statically, and the complicated special cases are hard to make accurately and completely[1], these facts make the effort questionable. Based on above work, the same and relative researchers tried to give a pure syntax-based Hoare Logic for OO programs in some later papers [38, 39]. However, the progress is not very much. It seems that the key

[1]We found an example obeying the restrictions but failed to be included in the definition. Although it is easy to remedy, it shows that managing the semantics in such a way is troublesome.

problems caused by OO programming, e.g. objects sharing and mutable object structures, are very hard to handle correctly and concisely in the classical Hoare Logic style with the syntactic substitution fashion.

A. Cavalcanti and D. Naumann made a significant contribution in the definition of a WP semantics for OO languages. In [9, 10], a simple OO language with subtyping, dynamic binding, but not sharing was studied. Supported by a statical typing environment, each command was associated with semantics as a predicate transformer. The notion of OO refinement is defined too. Based on the work, some inference rules for a model OO language ROOL were proved in [5]; a set of refactoring laws taken from [15] were presented formally in [11]. Further, paper [7] proposed a refinement algebra for OO programming. An extended paper [6] summarized these works. However, the semantics model for variables and fields of objects taken in these works is not based on the reference model. This start point departs from the essential features of the main stream OO languages, as well as the basic OO concepts, fundamentally. This departure makes the object sharing and updating hard to treat, if not impossible. A noticeable example can be found in [10], where only WP semantics of assignment with upcast can be defined. This shortage is due to value model and the absence of object references. Additionally, in [7] and [11], no refinement law related to references can be given. Further, in [11], some mistakes were made in attempting to encode the sharing-related refactoring in a non-reference semantic model. In summary, based on a value-based semantic model, it seems very hard, if not impossible, to formalize and verify many OO concerns interested in the practice.

On other directions, various specification and verification frameworks developed for OO programs, such as in ESC/JAVA [14], LOOP [8], JML [20, 18] and Spec# [4, 3], also utilize WP techniques as their basic tools. One common-seen usage is using some form of WP rules to generate verification conditions. However, these works focus mainly on useful, yet succinct notations for specifications of OO programs to support the verification requirements. As a result, none of the works focuses on the fundamental study of WP semantics. In addition, none puts the important mutable object structures as its basic consideration, as concluded in a survey [21] co-authored by developers of some of the frameworks.

With the rising and success of Separation Logic (SL), the main stream of the research on pointer-related and OO related programming has shifted to this line, and many problems related to OO programs have been reexamined. Some papers mentioned or used WP semantics based on some form of Separation Logic to attach problems in OO programming. For example, the work presented in [36] use a WP semantics to prove the soundness of their framework on Separation Logic and Implicit Dynamic Frames [42]. However, the WP semantics of command is defined as state transformers, but not predicate transformers, that is not abstract enough, thus is not very useful practically. There is no work on the fundamental research on the WP semantics yet.

The work presented here could also be thought as one attempt following the tide raised by Separation Logic. Compared to the existing works on WP semantics of OO programs, our semantics seems closer the realities of OO practice, and reflects the important OO features better. The facts come from that we adopt the underlying pure reference storage model, and design a logic with the help of the separation concept. The concepts and inference rules defined and proved in this paper can be seen as the additional evidences for this conclusion.

6.2. Concepts in OO Program Verifications

Behavioral subtyping is an important concept for OO program verification, and it always involves other crucial concepts like method specifications and object invariants. One earliest study on this topic is [25, 26], where a group of constraint rules are proposed to require that methods in a subtype preserve the behavior of corresponding methods in the supertype, and the invariant of the subtype implies the invariant of the supertype. However, as pointed by Leavens and Naumann in [22, 23], these constraints are too strong, so they offered a new definition based on the *nature specification refinement*, and proposed some general notations for specification and refinement. However, their concepts are defined based on the state transitions, thus are not high-level enough for supporting practical verification frameworks. Besides, in these works, the object invariant is treat as an always holding precondition of the methods. As we discussed in Section 4.3, this kind of requirements is not very adequate.

Specification refinement has also been investigated by many other researchers. For example, rCOS [19] defines refinement relationship by graph transformation, and Parkinson [37] defines refinement by a proof between specifications. In fact, all these definition follows the nature refinement order [22]. It seems that researchers have reached some consensus on specification refinement concept. We define the concept based on our WP semantics, and give also some sufficient conditions to support practical verifications. In addition, we take the field extension of subclasses into considerations in the definition and verification conditions.

In our knowledge, the most notable works on *object invariants* (and data invariants) are presented by Hoare [17], Barnett [2], Leino [24] and Müller [33]. The related techniques have been applied to various verification tools, such as ESC/JAVA [14], JML [20] and Spec# [4]. However, the definitions for object invariant in these works did not be given in some complete formal style. More important, the object invariant was treated all as a part of method specification. As discussed in Section 4.2, to define object invariant in this way has some inherent weakness. We give a different and better definition for the concept based on our WP semantics, which makes the invariant independent from any method specifications.

Verification frameworks for OO programs have induced much attentions, such as the works mentioned above. Because this is not the main topic of this paper, we will not discuss them in details here. As said in Section 5, the set of Hoare style inference rules and their soundness proof are given here as only an example for illustrating the usefulness of our WP semantics. Here we list only a set of basic rules. As a more useful set of rules, readers can consult paper [27]. In fact, we have proved also the soundness of this enriched set of rules.

7. Conclusion

Based on an OO version of Separation Logic, here we develop a WP semantics for a typical OO language with a large set of fundamental OO features, and prove that the semantics is both sound and complete. As far as we know, this is the first work on the completeness of such a semantics for OO languages with pure reference semantic model. In addition, some properties of the WP semantics are proposed and proved, especially the frame property of the logic with a detailed proof that is very important for local reasoning.

Based on the WP semantics, we investigate the behavioral subtyping and some other important concepts which are central for OO program verification. We introduction method specification and define the refinement relation on method specifications. We propose a new formal definitions for object invariants, and discuss why it is new and more adequate. We define also behavioral subtying relation based on the WP semantics, and provide some sufficient verification conditions for checking the relation for OO programs. As another application, we define (partial) correctness of OO programs with respect to their specification, and give a set of basic inference rules for reasoning OO programs with their soundness proofs.

Conducting a detailed comparison to existing works, e.g., [10, 12, 39, 25, 26, 22, 23], we might conclude that our WP semantics captures more essentials of object-orientation in an more adequate and useful way. And in addition, our formal treatments for the object invariant and behavioral subtyping is more natural and closer to the practice.

As for the future work, first, we will try to use the WP semantics and other concepts defined here to OO program verification area. We are working on a specification and verification framework for Java-like languages, with polymorphism, encapsulation and modular verification in mind. Second, we want to extend our specification refinement concept to data and program refinements, then study the relationship between programs/specifications at different abstract levels, that provides the possibility of programming from specifications or/and code generation.

Appendix A. OOSL: Some Details and Its Semantics

Here we give some details about OOSL. A complete treatment can be found in [29], including some properties of OOSL, and a careful comparison with some related works.

To represent the states of OO programs, we use three basic sets Name, Type and Ref. Because references in Ref are atomic, we assume two primitive functions:[2]

- $\mathsf{eqref} : \mathsf{Ref} \to \mathsf{Ref} \to$ bool, justifies whether two references are the same, i.e. for any $r_1, r_2 \in \mathsf{Ref}$, $\mathsf{eqref}(r_1, r_2)$ iff r_1 is same to r_2.
- $\mathsf{otype} : \mathsf{Ref} \to \mathsf{Type}$ decides the type of the object referred by some reference. We define $\mathsf{otype}(\mathsf{rtrue}) = \mathsf{otype}(\mathsf{rfalse}) = \mathtt{Bool}$, and $\mathsf{otype}(\mathsf{rnull}) = \mathtt{Null}$.

A program state $s = (\sigma, O) \in$ State consists of a store and a heap. An element of O is a pair (r, f), where f is an abstraction of some object o pointed by r, a function from fields of o to values.[3] For domain of O, we refer to either a subset of Ref associated with objects, or a subset of $\mathsf{Ref} \times \mathsf{Name}$ associated with values. We use $\mathsf{dom}\, O$ to denote the domain of O, and define $\mathsf{dom}_2\, O \mathrel{\hat=} \{(r, a) \mid r \in \mathsf{dom}\, O, a \in \mathsf{dom}\, O(r)\}$ for the second case.

For the program states, we define the well-typedness as follows.

Definition 12 (Well-Typed States). State $s = (\sigma, O)$ is well-typed if both its store σ and heap O are well-typed, where store σ is well-typed if $\forall v \in \mathsf{dom}\, \sigma \cdot \mathsf{otype}(\sigma(v)) <: \mathsf{dtype}(v)$; and heap O is well-typed if the following two conditions hold:

- $\forall (r, a) \in \mathsf{dom}_2\, O \cdot a \in \mathsf{fields}(\mathsf{otype}(r)) \wedge \mathsf{otype}(O(r)(a)) <: \mathsf{fdtypes}(\mathsf{otype}(r))(a)$, and
- $\forall r \in \mathsf{dom}\, O \cdot \mathsf{fields}(\mathsf{otype}(r)) = \emptyset \vee (\mathsf{fields}(\mathsf{otype}(r)) \cap \mathsf{dom}\, O(r) \neq \emptyset)$. □

Clearly, a well-typed store has all its variables taking values of the valid types. On the other hand, a well-typed heap requires that: 1) all fields in O are valid according to their objects, and hold values of valid types; and 2) for a non-empty object (according to its type), only when at

[2]One possible implementation, for example, is to define a reference as a pair (t, id) where $t \in \mathsf{Type}$ and $id \in \mathbf{N}$, and define eqref as the pair equality, and $\mathsf{otype}(r) = r.first$.

[3]Please pay attention that $\mathsf{Ref} \rightharpoonup_{\mathsf{fin}} \mathsf{Name} \rightharpoonup_{\mathsf{fin}} \mathsf{Ref}$ is very different from $\mathsf{Ref} \times \mathsf{Name} \rightharpoonup_{\mathsf{fin}} \mathsf{Ref}$. Informally speaking, the former is a map from references to objects, while the latter is a map from object fields to field values. So objects have no direct presentations in the latter, especially empty objects.

$$
\begin{array}{rl}
\text{[I-FALSE]}\ \mathcal{M}_{\mathcal{J}}(\mathtt{false}) = \emptyset & \text{[I-TRUE]}\ \mathcal{M}_{\mathcal{J}}(\mathtt{true}) = \text{State} \\
\text{[I-LOOKUP]}\ \mathcal{M}_{\mathcal{J}}(v = r) = \{(\sigma, O) \mid \sigma(v) = r\} & \text{[I-REF-EQ]}\ \mathcal{M}(r_1 = r_2) = \text{State if } \mathsf{eqref}(r_1, r_2),\ \emptyset \text{ else} \\
\text{[I-REF-TP]}\ \mathcal{M}(r : T) = \text{State if } \mathsf{otype}(r) = T,\ \emptyset \text{ else} & \text{[I-REF-STP]}\ \mathcal{M}(r <: T) = \text{State if } \mathsf{otype}(r) <: T,\ \emptyset \text{ else} \\
\text{[I-EMPTY]}\ \mathcal{M}_{\mathcal{J}}(\mathbf{emp}) = \{(\sigma, \emptyset)\} & \text{[I-SINGLE]}\ \mathcal{M}_{\mathcal{J}}(r_1.a \mapsto r_2) = \{(\sigma, \{(r_1, a, r_2)\})\} \\
\text{[I-OBJ]}\ \mathcal{M}_{\mathcal{J}}(\mathsf{obj}(r, T)) = \{(\sigma, O) \mid \mathsf{dom}\, O = \{r\} \wedge \mathsf{dom}\,(O(r)) = \mathsf{fields}(\mathsf{otype}(r))\} & \\
\text{[I-APP]}\ \mathcal{M}_{\mathcal{J}}(p(\overline{r})) = \mathcal{J}(p)(\overline{r}) & \\
\text{[I-NEG]}\ \mathcal{M}_{\mathcal{J}}(\neg\psi) = \text{State} \setminus \mathcal{M}_{\mathcal{J}}(\psi) & \text{[I-OR]}\ \mathcal{M}_{\mathcal{J}}(\psi_1 \vee \psi_2) = \mathcal{M}_{\mathcal{J}}(\psi_1) \cup \mathcal{M}_{\mathcal{J}}(\psi_2) \\
\text{[I-S-CONJ]}\ \mathcal{M}_{\mathcal{J}}(\psi_1 * \psi_2) = \{(\sigma, O) \mid \exists O_1, O_2 \cdot O_1 * O_2 = O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{J}}(\psi_1) \wedge (\sigma, O_2) \in \mathcal{M}_{\mathcal{J}}(\psi_2)\} & \\
\text{[I-S-IMPLY]}\ \mathcal{M}_{\mathcal{J}}(\psi_1 -\!\!* \psi_2) = \{(\sigma, O) \mid \forall O_1 \cdot O_1 \perp O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{J}}(\psi_1) \text{ implies } (\sigma, O_1 * O) \in \mathcal{M}_{\mathcal{J}}(\psi_2)\} & \\
\text{[I-EX]}\ \mathcal{M}_{\mathcal{J}}(\exists r \cdot \psi) = \{(\sigma, O) \mid \exists r \in \mathsf{Ref} \cdot (\sigma, O) \in \mathcal{M}_{\mathcal{J}}(\psi)\} &
\end{array}
$$

Figure A.7: Semantic for OOSL wrt. the least fixed point model $\mathcal{J}$ of the given logic environment

least one of its fields is in O, we can say the object is in O. Thus we can identify empty objects in any heap. We will only consider well-typed states in our study.

We define a special overriding operator $\oplus$ on Heap:

$$
(O_1 \oplus O_2)(r) \mathrel{\hat{=}} \begin{cases} O_1(r) \oplus O_2(r) & \text{if } r \in \mathsf{dom}\, O_2 \\ O_1(r) & \text{otherwise} \end{cases}
$$

The $\oplus$ operator on the right hand side is the standard function overriding. Thus, for heap O_1, $O_1 \oplus \{(r, a, r')\}$ gives a new heap, where the value for only one field (the value for a) of the object pointed by r is modified (to the value denoted by r').

We use $O_1 \perp O_2$ to indicate that O_1 and O_2 are separated from each other:

$$
O_1 \perp O_2 \mathrel{\hat{=}} \forall r \in \mathsf{dom}\, O_1 \cap \mathsf{dom}\, O_2 \cdot (O_1(r) \neq \emptyset \wedge O_2(r) \neq \emptyset \wedge \mathsf{dom}\,(O_1(r)) \cap \mathsf{dom}\,(O_2(r)) = \emptyset).
$$

If a reference, to some object o, is in both domains of two heaps O_1 and O_2, then each of O_1 and O_2 must contain a non-empty subset o's fields (the well-typedness guarantees this), and the two subsets must be disjoint. This means that we can separate fields of a non-empty object into different heaps, but not an empty object. With this definition, we have $O \perp \emptyset$ for any O, as well as $\emptyset \perp \emptyset$. When $O_1 \perp O_2$, we will use $O_1 * O_2$ for the union $O_1 \cup O_2$.

The storage model defined above, with the definition for the separation concept, gives us both an object view and a field view for the heaps. With this model, we can correctly handle the whole objects and their fields.

To define the semantics for OOSL, we need to have a careful treatment about the *user-defined assertions*. Because we allow recursive definitions (either self-recursion or mutual recursion), any reasonable definition for their semantics must involve some fixed point. We record all the definitions in a *Logic Environment* Λ:

$$
\Lambda ::= \varepsilon \mid p(\overline{r}) \doteq \psi, \Lambda
$$

Here ε denotes the empty environment.

As the well-formedness, body ψ of any definition in Λ cannot use symbols without defined in Λ. Further, we require that Λ is *finite* and any body predicate ψ in it is *syntactically monotone*[4].

[4]For definition $p(\overline{r}) \doteq \psi$, every symbol occurred in ψ must lie under even number of negations.

Under these conditions, the *least fix-point model* for a given Λ exists by Tarski's theorem, then we name it as $\mathcal{J}$. Based on this model, we define semantics of assertions by a semantic function $\mathcal{M}_\Lambda : \Psi \to \mathbb{P}(\text{State})$ by rules listed in **Fig. A.7**, where subscript Λ is omitted as default.

With this semantics, we define that an assertion holds on a given state as:

$$(\sigma, O) \models \psi \qquad \text{iff} \qquad (\sigma, O) \in \mathcal{M}(\psi).$$

Appendix A.1. Properties and Inference Rules

The semantics defined above have many good properties. We give some of them in this subsection.

Lemma 4. *New predicate can be safely appended to Λ, without changing the meaning of existing symbols in Λ. Formally, if $\Lambda' = (\Lambda, p(\overline{r}) \doteq \psi)$ is a well-formed logic environment, where p is not defined in Λ, we have for every symbol q defined in Λ:*

$$\mathcal{J}_\Lambda(q) = \mathcal{J}_{\Lambda'}(q).$$

Proof. By the definition of standard model $\mathcal{J}$. □

By this lemma, we can easily get:

Lemma 5 (Extending and Shrinking)**.** *Given a logic environment Λ:*

1. *We can safely append some new definitions to it, without changing semantics of symbols defined in Λ, providing the extension is a well-formed logic environment;*
2. *if symbols $\overline{p}$ defined in Λ are not mentioned in other definitions in Λ, then we can safely remove them, without changing the meaning of the rest symbols in Λ.*

We have following results for store extension and shrinking:

Lemma 6 (Stack Extending and Shrinking)**.** *Suppose $(\sigma, O) \models \psi$, we have:*

1. *if $\mathsf{dom}\,\sigma' \cap \mathsf{dom}\,\sigma = \emptyset$, then $(\sigma \cup \sigma', O) \models \psi$;*
2. *if ψ does not contain variables in σ', then $(\sigma - \sigma', O) \models \psi$. Here $\sigma - \sigma'$ denotes the state $\{(x, r) \in \sigma \mid x \notin \mathsf{dom}\,\sigma'\}$*

Lemma 7 (Variable Substitution)**.** $(\sigma, O) \models \psi[e/x]$, *if and only if* $(\sigma \oplus \{x \mapsto \sigma e\}, O) \models \psi$.

Lemma 8. *Suppose $a_1, a_2, ..., a_k$ are all fields of type T, then we have:*

$$\mathsf{obj}(r, T) \Leftrightarrow r.a_1 \mapsto - * r.a_2 \mapsto - * \ldots * r.a_k \mapsto -$$

This law denotes the transformation (correspondence) between the object based viewpoint and fields based viewpoint.

Lemma 9. $\mathsf{obj}(r_1, -) * \mathsf{obj}(r_2, -) \Rightarrow r_1 \neq r_2$.

Lemma 10. *We can prove many laws, such as:*

$$\begin{array}{ll} \mathbf{emp} * \psi \Leftrightarrow \psi & \psi_1 \mathrel{-\!\!*} (\psi_2 \wedge \psi_3) \Leftrightarrow (\psi_1 \mathrel{-\!\!*} \psi_2) \wedge (\psi_1 \mathrel{-\!\!*} \psi_3) \\ \psi_1 * (\psi_1 \mathrel{-\!\!*} \psi_2) \Leftrightarrow \psi_2 & \psi_1 \mathrel{-\!\!*} \psi_2 \mathrel{-\!\!*} \psi_3 \Leftrightarrow (\psi_1 * \psi_2) \mathrel{-\!\!*} \psi_3 \end{array}$$

Proof. We prove the last statement here. Note that $\psi_1 \mathrel{-\!\!*} \psi_2 \mathrel{-\!\!*} \psi_3$ is $\psi_1 \mathrel{-\!\!*} (\psi_2 \mathrel{-\!\!*} \psi_3)$.

$\Rightarrow$: Assume $(\sigma, O) \models \psi_1 \mathrel{-\!\!*} (\psi_2 \mathrel{-\!\!*} \psi_3)$. Take any O' such that $O' \perp O$ and $(\sigma, O') \models \psi_1 * \psi_2$, by the definition of $*$, there exist O_1 and O_2 such that $O' = O_1 * O_2$, and $(\sigma, O_1) \models \psi_1$, and $(\sigma, O_2) \models \psi_2$. Because $O_1 \perp O_2 * O$ and the assumption, we know that $(\sigma, O_1 * O) \models \psi_2 \mathrel{-\!\!*} \psi_3$. From this fact, and $(\sigma, O_2) \models \psi_2$ and $O_2 \perp O_1 * O$, we have $(\sigma, O_1 * O_2 * O) \models \psi_3$. This is exactly $(\sigma, O' * O) \models \psi_3$, thus we have the "$\Rightarrow$" proved.

$\Leftarrow$: Suppose $(\sigma, O) \models (\psi_1 * \psi_2) \mathrel{-\!\!*} \psi_3$. Take any O_1 such that $O_1 \perp O$ and $(\sigma, O_1) \models \psi_1$, then take any O_2 such that $O_2 \perp O_1 * O$ and $(\sigma, O_2) \models \psi_2$, now we need to prove that $(\sigma, O_1 * O_2 * O) \models \psi_3$. Because $O_1 * O_2 \perp O$ and $(\sigma, O_1 * O_2) \models \psi_1 * \psi_2$, we have the result immediately. □

Many propositions in Separation Logic also hold in OOSL. For example, laws (i.e., axiom schemata) given in Section 3 of [41] are all valid in OOSL. Some more laws can be proved which are also valid in Separation Logic, such as (ref. [43]):

$$\begin{array}{rcl@{\qquad\qquad}rcl} (p * q) * r & \Leftrightarrow & p * (q * r) & p * q & \Leftrightarrow & p * (p \mathrel{-\!\!*} (p * q)) \\ \mathbf{emp} & \Rightarrow & p \mathrel{-\!\!*} p & p \mathrel{-\!\!*} q & \Leftrightarrow & p \mathrel{-\!\!*} (p * (p \mathrel{-\!\!*} q)) \end{array}$$

Intuitively, there are close connection between OOSL defined here and the Separation Logic. If we treat every tuple (r, a) as an address of memory cell, and define a suitable address transformation for the memory layout, then we may map the storage model of our logic to the storage model of Separation Logic. So, we conjecture that every proposition holding in Separation Logic, when it does not involve in address arithmetic, will hold in OO Separation Logic. We will investigate the relation between Separation Logic and OOSL in future.

Similar to Separation Logic, we can define the *pure*, *intuitionistic*, *strictly-exact* and *domain-exact* assertions. We find another important concept as follows.

Definition 13 (Separated Assertions). Two assertions ψ and ψ' are *separated* from each other, *iff* for all stores σ and Opools O, O', $(\sigma, O) \models \psi$ and $(\sigma, O') \models \psi'$ implies $O \perp O'$.

Clearly, the separation for assertions is symmetric.

Lemma 11. *$r_1.a \mapsto -$ and $r_2.b \mapsto -$ are separated, provided that $r_1 \neq r_2$, or a and b are different field names.*

Note that "$r_1 \neq r_2$" can not be determined statically, but "a and b are different field names" can. Suppose we have a *Node* class with fields *value* and *next*. For a reference $r : Node$, we know *r.value* $\mapsto -$ and *r.next* $\mapsto -$ are separated. This concept can also be defined in Separation Logic. However, due to the absence of named fields, the fact that two assertions are separated on the syntactic level is not common, except some special cases, such as $x \mapsto -$ and $x + 1 \mapsto -$ (however, for $x + n$ when n is a variable we can say nothing).

Lemma 12 (Separated Assertions and Heaps)**.** *Suppose ψ_1 and ψ_2 are separated.*

(1). *If $(\sigma, O_1) \models \psi_1$ and $(\sigma, O_2) \models \psi_2$, then $(\sigma, O_1 * O_2) \models \psi_1 * \psi_2$.*

(2). *If $(\sigma, O) \models \psi_1 * \psi_2$, there exists an unique partition of $O = O_1 * O_2$ such that $(\sigma, O_1) \models \psi_1$ and $(\sigma, O_2) \models \psi_2$.*

Proof. Result (1) is trivially true by the separation of ψ_1 and ψ_2. We prove here only (2).

Suppose $(\sigma, O) \models \psi_1 * \psi_2$ and we have two different partitions $O = O_1 * O_2$ and $O = O'_1 * O'_2$ such that $(\sigma, O_1) \models \psi_1$ and $(\sigma, O_2) \models \psi_2$, also $(\sigma, O'_1) \models \psi_1$ and $(\sigma, O'_2) \models \psi_2$. Because $O_1 \neq$

O'_1, without lose the generality, we can suppose $(\mathsf{dom}_2\, O_1 \setminus \mathsf{dom}_2\, O'_1) \neq \emptyset$. Then we have $(\mathsf{dom}_2\, O_1 \setminus \mathsf{dom}_2\, O'_1) \subseteq \mathsf{dom}_2\, O_1$ and $(\mathsf{dom}_2\, O_1 \setminus \mathsf{dom}_2\, O'_1) \subseteq \mathsf{dom}_2\, O'_2$, thus $O_1 \not\perp O'_2$. This is not possible because $(\sigma, O_1) \models \psi_1$, $(\sigma, O'_2) \models \psi_2$ and ψ_1 and ψ_2 are separated assertions. $\square$

Lemma 13. *Suppose ψ_2 and ψ_3 are two assertions separated from each other, ψ_1 is separated from both of ψ_2 and ψ_3 iff ψ_1 is separated from $\psi_2 * \psi_3$.*

Proof. The proof is as follows:

- $\Rightarrow$: Take any σ, O_1 and O such that $(\sigma, O_1) \models \psi_1$ and $(\sigma, O) \models \psi_2 * \psi_3$, and take any partition $O = O_2 * O_3$ such that $(\sigma, O_2) \models \psi_2$ and $(\sigma, O_3) \models \psi_3$. Because ψ_1 is separated from both ψ_2 and ψ_3, then $O_1 \perp O_2$ and $O_1 \perp O_3$. Thus $O_1 \perp (O_2 * O_3) = O$, which tells us ψ_1 is separated from $\psi_2 * \psi_3$.

- $\Leftarrow$: Suppose $(\sigma, O_1) \models \psi_1$ and $(\sigma, O_2) \models \psi_2$. For any O_3 such that $(\sigma, O_3) \models \psi_3$, because ψ_2 and ψ_3 are separated, then $O_2 \perp O_3$ and $(\sigma, O_2 * O_3) \models \psi_2 * \psi_3$. Because ψ_1 is separated from $\psi_2 * \psi_3$, then $O_1 \perp (O_2 * O_3)$, and then $O_1 \perp O_2$. For the arbitrary choices of O_1 and O_2 , we conclude that ψ_1 is separated from ψ_2. For ψ_1 and ψ_3, the proof is similar. $\square$

Theorem 10. *For any assertions ψ_1, ψ_2, ψ_3, if ψ_1 and ψ_2 are separated from each other, then $\psi_1 * (\psi_2 \mathrel{-\!\!*} \psi_3) \Leftrightarrow \psi_2 \mathrel{-\!\!*} (\psi_1 * \psi_3)$.*

Proof. The proof is as follows:

- $\Rightarrow$: For any σ and O such that $(\sigma, O) \models \psi_1 * (\psi_2 \mathrel{-\!\!*} \psi_3)$, there exist O_1, O_2, such that $O_1 * O_2 = O$, $(\sigma, O_1) \models \psi_1$, and $(\sigma, O_2) \models \psi_2 \mathrel{-\!\!*} \psi_3$. By the definition of$\mathrel{-\!\!*}$, for any O_3 satisfying $O_2 \perp O_3$, we have
$$(\sigma, O_3) \models \psi_2 \text{ implies } (\sigma, O_2 * O_3) \models \psi_3.$$
  Because ψ_1 and ψ_2 are separated, then by **Lemma 12**,
$$(\sigma, O_3) \models \psi_2 \text{ implies } (\sigma, O_1 * O_2 * O_3) \models \psi_1 * \psi_3.$$
  This is $(\sigma, O) \models \psi_2 \mathrel{-\!\!*} (\psi_1 * \psi_3)$.

- $\Leftarrow$: For any σ and O that $(\sigma, O) \models \psi_2 \mathrel{-\!\!*} (\psi_1 * \psi_3)$, for any O_1 that $O_1 \perp O$, if $(\sigma, O_1) \models \psi_2$, then $(\sigma, O_1 * O) \models \psi_1 * \psi_3$. Now we fix this O_1. From $(\sigma, O_1 * O) \models \psi_1 * \psi_3$ we know there exist O_2 and O'_3 such that $O_2 \perp O'_3$, $O_2 * O'_3 = O_1 * O$, $(\sigma, O_2) \models \psi_1$ and $(\sigma, O'_3) \models \psi_3$. Because ψ_1, ψ_2 are separated, then $O_2 \perp O_1$. Thus $O'_3 = O_1 * O_3$ for some O_3. Now we have
$$(\sigma, O_2) \models \psi_1,\ (\sigma, O_1) \models \psi_2, \text{ and } (\sigma, O_1 * O_3) \models \psi_3.$$
  Then we have $(\sigma, O_3) \models \psi_2 \mathrel{-\!\!*} \psi_3$, because the choice of O_1 needs no extra restriction. Thus $(\sigma, O) \models \psi_1 * (\psi_2 \mathrel{-\!\!*} \psi_3)$, because $O = O_2 * O_3$. $\square$

The concept of *separated assertions* is very useful in reasoning OO programs. Taking the *Node* class above as an example, it allows us to combine relative fields of a *Node* object together:

$$\begin{array}{rl} & r_1.value \mapsto - * (r_2.value \mapsto - \mathrel{-\!\!*} r_1.next \mapsto -) \\ \Leftrightarrow & r_2.value \mapsto - \mathrel{-\!\!*} (r_1.value \mapsto - * r_1.next \mapsto -) \end{array}$$

Now we discuss some expressiveness and extension issues about OOSL.

$$\underline{\text{[O-BASIC-EXP]}}\quad \{\!|\mathbf{true}|\!\}_{(\sigma,O)} = \mathsf{rtrue},\quad \{\!|\mathbf{false}|\!\}_{(\sigma,O)} = \mathsf{rfalse},\quad \{\!|\mathbf{null}|\!\}_{(\sigma,O)} = \mathsf{rnull},\quad \{\!|v|\!\}_{(\sigma,O)} = \sigma v$$

$$\underline{\text{[O-BOOL-EXP]}}\quad \{\!|e_1 = e_2|\!\}_{(\sigma,O)} = \begin{cases} \mathsf{rtrue}, & \text{if } \{\!|e_1|\!\}_{(\sigma,O)} = \{\!|e_2|\!\}_{(\sigma,O)} \\ \mathsf{rfalse}, & \text{else} \end{cases} \qquad \frac{\{\!|b|\!\}_{(\sigma,O)} = \mathsf{rtrue}}{\{\!|\neg b|\!\}_{(\sigma,O)} = \mathsf{rfalse}} \qquad \frac{\{\!|b|\!\}_{(\sigma,O)} = \mathsf{rfalse}}{\{\!|\neg b|\!\}_{(\sigma,O)} = \mathsf{rtrue}}$$

Figure B.8: Expression Evaluation

As presented above, we define a power assertion language for OOSL, especially the user-defined predicates are treated formally in the framework, which notably enhance the expressiveness of OOSL. With OOSL, We can specify and infer recursive data structures, e.g., lists, trees, etc., and some important properties between objects, such as accessibility, dangling and so on. Since our logic adopts the classical semantics, it is more expressive than its intuitionistic cousin, e.g., what defined and used in [35]. We can use OOSL to describe the program state precisely, especially the Opool, i.e., what is in or is not in an Opool.

On the other hand, the primitive assertions of OOSL are very simple and limited, especially we have only Boolean type here. Due to the limited mathematical basics, we cannot describe quantitative relation or more complicated mathematical concepts with OOSL. But it is not difficult to extend OOSL to support these concepts. For example, if we want to support integer arithmetic in OOSL, we should

- add some primitive assertions about integer,
- expand user-defined predicates with integer arguments,
- expand quantifiers $\exists$ and $\forall$ to support integer,
- define semantics for new adding assertions.

After these modifications, we can describe and infer properties involving integers with OOSL. In fact, we can combine OOSL and other mathematical theories freely, such as theories about sequences and trees, since they are orthogonal.

Appendix B. Soundness and Completeness of the WP Semantics

In this Appendix we prove the soundness and completeness theorems given in Section 3.6 with respect to an operational semantics of μJava defined below.

Appendix B.1. Operational Semantics of μJava

To prove the sound and complete theorems for the WP semantics defined in Section 3.2, we give first an operational semantics for μJava, which will be used as a base for the proofs.

As said before, μJava takes a pure reference semantics for variables and fields of objects, thus, here a primitive type is also thought as an object type. The semantics of generic and Boolean expressions e and b in current state (σ, O) are represented as $\{\!|e|\!\}_{(\sigma,O)}$ or $\{\!|b|\!\}_{(\sigma,O)}$, respectively. The evaluation rules for expressions are given in Figure B.8. For a well-typed program, the evaluation of v always makes sense. For the Boolean connectors, we give here only semantics to $\neg$. The semantics for operators $\vee$, $\wedge$, etc. is standard. Because the restricted form of expressions, semantics of expressions depend only on the store but not the heap, thus we will use σe or σb as the abbreviations for $\{\!|e|\!\}_{(\sigma,O)}$ or $\{\!|b|\!\}_{(\sigma,O)}$ sometimes below.

$$[\underline{\text{O-SEQ}}]\ \frac{\langle c_1,(\sigma,O)\rangle \rightsquigarrow^* (\sigma',O'),\ \ \langle c_2,(\sigma',O')\rangle \rightsquigarrow^* \theta}{\langle c_1;c_2,(\sigma,O)\rangle \rightsquigarrow^* \theta} \qquad \frac{\langle c_1,(\sigma,O)\rangle \rightsquigarrow^* \mathsf{abort}}{\langle c_1;c_2,(\sigma,O)\rangle \rightsquigarrow^* \mathsf{abort}}$$

$$[\underline{\text{O-ITER}}]\ \frac{\sigma b = \mathsf{rtrue},\ \ \langle c;\texttt{while}\ \ b\ c,(\sigma,O)\rangle \rightsquigarrow^* \theta}{\langle \texttt{while}\ \ b\ c,(\sigma,O)\rangle \rightsquigarrow^* \theta} \qquad \frac{\sigma b = \mathsf{rfalse}}{\langle \texttt{while}\ \ b\ c,(\sigma,O)\rangle \rightsquigarrow (\sigma,O)}$$

$$[\underline{\text{O-COND}}]\ \frac{\sigma b = \mathsf{rtrue},\ \ \langle c_1,(\sigma,O)\rangle \rightsquigarrow^* \theta}{\langle \mathbf{if}\ b\ c_1\ \mathbf{else}\ c_2,(\sigma,O)\rangle \rightsquigarrow^* \theta} \quad \frac{\sigma b = \mathsf{rfalse},\ \ \langle c_2,(\sigma,O)\rangle \rightsquigarrow^* \theta}{\langle \mathbf{if}\ b\ c_1\ \mathbf{else}\ c_2,(\sigma,O)\rangle \rightsquigarrow^* \theta} \quad [\underline{\text{O-SKIP}}]\ \frac{}{\langle \texttt{skip},(\sigma,O)\rangle \rightsquigarrow (\sigma,O)}$$

$$[\underline{\text{O-ASN}}]\ \frac{}{\langle x := e,(\sigma,O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto \sigma e\},O)} \qquad [\underline{\text{O-RET}}]\ \frac{}{\langle \texttt{return}\ \ e,(\sigma,O)\rangle \rightsquigarrow (\sigma \oplus \{\mathsf{res} \mapsto \sigma e\},O)}$$

$$[\underline{\text{O-MUT}}]\ \frac{(\sigma v,a) \in \mathsf{dom}_2\ O}{\langle v.a := e,(\sigma,O)\rangle \rightsquigarrow (\sigma, O \oplus \{(\sigma v,a,\sigma e)\})} \qquad \frac{(\sigma v,a) \notin \mathsf{dom}_2\ O}{\langle v.a := e,(\sigma,O)\rangle \rightsquigarrow \mathsf{abort}}$$

$$[\underline{\text{O-LOOKUP}}]\ \frac{(\sigma v,a) \in \mathsf{dom}_2\ O}{\langle x := v.a,(\sigma,O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto O(\sigma v)(a)\},O)} \quad \frac{(\sigma v,a) \notin \mathsf{dom}_2\ O}{\langle x := v.a,(\sigma,O)\rangle \rightsquigarrow \mathsf{abort}}$$

$$[\underline{\text{O-CAST}}]\ \frac{\mathsf{otype}(\sigma v) <: N}{\langle x := (N)v,(\sigma,O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto \sigma v\},O)} \qquad \frac{\mathsf{otype}(\sigma v) \not<: N}{\langle x := (N)v,(\sigma,O)\rangle \rightsquigarrow \mathsf{abort}}$$

$$[\underline{\text{O-INV}}]\ \frac{\begin{array}{c}\mathsf{otype}(\sigma v) = C,\quad \Theta,C,m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y};c\}\\ \langle c,(\{\texttt{this} \mapsto \sigma v, \overline{z \mapsto \sigma e}, \overline{y \mapsto \mathsf{nil}}, \mathsf{res} \mapsto \mathsf{nil}\},O)\rangle \rightsquigarrow^* (\sigma',O')\end{array}}{\langle x := v.m(\overline{e}),(\sigma,O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto \sigma'\mathsf{res}\},O')}$$

$$[\underline{\text{O-NEW}}]\ \frac{\begin{array}{c}\mathsf{fdtypes}(C) = \{\overline{a:T}\},\quad \Theta,C,C \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y};c\}\\ \langle c,(\{\texttt{this} \mapsto r, \overline{z \mapsto \sigma e}, \overline{y \mapsto \mathsf{nil}}\}, O \oplus \{(r,\{\overline{(a,\mathsf{nil})}\})\})\rangle \rightsquigarrow^* (\sigma',O')\end{array}}{\langle x := \texttt{new}\ \ C(\overline{e}),(\sigma,O)\rangle \rightsquigarrow (\sigma \oplus \{x \mapsto r\},O')}\ r \notin \mathsf{dom}\,O$$

Figure B.9: Operational Semantics for μJava

For describing the return value transferring, we assume that every store contains an internal-variable res for recording the return value of recently invoked method. Because μJava is sequential, we just need one res to specify the return value at any time. We assume that a res always takes the correct type with its method.

The operational semantics is defined as a mapping from configurations to configurations. A configuration is either a tuple $\langle c, s\rangle$ consisting of a program text and a state, or a terminated state $s = (\sigma, O)$, represents that the execution of a (piece of) program has completed successfully. The semantics is defined as a transition relation $\rightsquigarrow$:

$$\begin{array}{rcl} \text{Configuration} & \widehat{=} & (\text{Command} \times \text{State}) \cup \text{State} \\ \rightsquigarrow & \in & \text{Configuration} \rightharpoonup \text{Configuration} \cup \{\mathsf{abort}\} \end{array}$$

Here Command is the set of valid program texts, and abort is a special symbol to represent that the program goes wrong in execution, because of memory faults, wrong type casts, etc. We define $\rightsquigarrow^*$ the finite transition closure of $\rightsquigarrow$. The operational semantics rules for commands are given in Fig. B.9, where we take the big-step style. With the help of Γ, here we can get rid of many side conditions that can be guaranteed by the typing.

Rules [O-SEQ], [O-COND], and [O-ITER] for structural commands are simple where we use θ to represent either a terminal state or the abort. One can see that abort stops the execution, and it propagates further until the whole program stops.

Command skip changes nothing. The plain assignment $x := e$ is independent of Opool, thus affects only the store, as described by rule [O-ASN]. Both *lookup* and *mutation* look into the Opool. They go abort when dereferencing a field out of current Opool, that covers the cases where variable v has a rnull value, because then $(\sigma v, a)$ is out of any Opool. Rules [O-MUT] and [O-LOOKUP] capture the behaviors of these two commands.

Rule [O-CAST] shows that command $x := (N)v$ needs to check whether $(N)v$ is valid at run time. If it is not the case, the execution fails and reveals a wrong downcasting. The upcasting in the well-typed program is always allowed by the rule.

Rule [O-INV] defines the semantics for method invocations. It captures the dynamic binding feature of OO programs. As seen, the method invoked is determined by the type of the object pointed by variable v at run time. Body command c of method m executes from a new store created locally, with parameters initialized by real arguments, and local variables initialized to the special nil values (represented as nil in the rule) according to their types, where we assume rfalse for `Bool` and rnull for all the class types. We bind `this` to the object referred by v before the execution of c, and this binding keeps unchanged all the course. The internal variable res is initialized also to nil value. As said before, we assume all field references for the current object are decorated with `this`. Rule [O-RET] assigns the return value to res. When m returns, x is updated by value of res taken from the final state of the local store. We should have another rule for the case that the execution of method body is stuck resulting an abort. It takes the similar form as [O-INV] but propagates abort, thus we omit it omitted here.

Command $x :=$ `new` $C(\overline{e})$ creates a new object of class C, initiates its fields (including inherited ones) with command c, and lets x refer to the object. [O-NEW] asks r to be a fresh reference, whose selection is non-deterministic. $\{(r, \{\overline{(a, \mathsf{nil})}\})\}$ stands for every field value of r is nil.

Obviously, a program might fail to terminate for falling into an infinite iteration, or infinite method call chain. In these cases, the deduction defined by these rules cannot terminate either.

We have the following lemmas for store extension and shrink.

Lemma 14. (1) *Suppose* $\langle c, (\sigma_1, O_1)\rangle \rightsquigarrow^* (\sigma_2, O_2)$, *and* $\mathsf{dom}\,\sigma \cap \mathsf{dom}\,\sigma_1 = \emptyset$, *then* $\langle c, (\sigma_1 \cup \sigma, O_1)\rangle \rightsquigarrow^* (\sigma_2 \cup \sigma, O_2)$. (2) *Suppose* $\langle c, (\sigma_1, O_1)\rangle \rightsquigarrow^* (\sigma_2, O_2)$, *and c does not contain variables in* $\mathsf{dom}\,\sigma$, *then* $\langle c, (\sigma_1 - \sigma, O_1)\rangle \rightsquigarrow^* (\sigma_2 - \sigma, O_2)$. *Here* $\sigma_1 - \sigma$ *denotes the function by restricting* σ_1 *to the domain* $\mathsf{dom}\,\sigma_1 - \mathsf{dom}\,\sigma$. □

Now we are ready to prove the soundness and completeness theorems.

Appendix B.2. Soundness Theorem

We prove first the soundness theorem (**Theorem 4** in Section 3.6). Remember Ψ is the space of legal predicates and **COM** the space of legal commands. We will write $[\![c]\!]$ instead of $[\![\Gamma, C, m \vdash c : \mathbf{com}]\!]$ when it makes no confusion.

Proof. We prove that, for any $\psi, \psi' \in \Psi$ and $c \in$ **COM** satisfying $[\![\Gamma, C, m \vdash c : \mathbf{com}]\!]\psi = \psi'$, then for any pair of states (σ, O) and (σ', O'), if $(\sigma, O) \models \psi'$ and $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, then $(\sigma', O') \models \psi$. We prove it by induction on the structure of commands. We adopt some notations used in defining the WP semantics, and assume always ψ the postcondition.

Sequential Composition, "c_1; c_2": Suppose (σ, O) satisfy $[\![c_1; c_2]\!]\psi$, $[\![c_1]\!] = f_1$, and $[\![c_2]\!] = f_2$, thus we have $(\sigma, O) \models f_1(f_2(\psi))$. Assume that $\langle c_1, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, and $\langle c_2, (\sigma', O')\rangle \rightsquigarrow^* (\sigma'', O'')$. By induction hypothesis, $(\sigma', O') \models f_2(\psi)$, and also $(\sigma'', O'') \models \psi$.

Condition, "`if` b c_1 `else` c_2": Suppose (σ, O) satisfy the precondition, that is, $(\sigma, O) \models (b \Rightarrow f_1(\psi)) \wedge (\neg b \Rightarrow f_2(\psi))$. Further, suppose $\langle c_1, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, and $\langle c_2, (\sigma, O)\rangle \rightsquigarrow^* (\sigma'', O'')$. Then, if $\{\!|b|\!\}_{(\sigma,O)} = \mathsf{rtrue}$, we have $(\sigma, O) \models f_1(\psi)$. By induction hypothesis, we have $(\sigma', O') \models \psi$, so the conclusion holds. The case when $\{\!|b|\!\}_{(\sigma,O)} = \mathsf{rfalse}$ is similar.

Iteration, "$\texttt{while}\ b\ c$": Because $\texttt{while}\ b\ c$ is equivalence to $\texttt{if}\ b\ (c; \texttt{while}\ b\ c)\texttt{else skip}$, by the definition of $[\![\texttt{while}\ \ b\ c]\!]$ and proof for Condition, the conclusion holds.

Skip: The proof is trivial.

Assignment, "$x\ :=\ e$": Suppose (σ, O) satisfy the precondition, i.e., $(\sigma, O) \models \psi[e/x]$. By operational semantics rule [O-ASN], we have $(\sigma', O') = (\sigma \oplus \{x \mapsto \sigma e\}, O)$. By **Lemma 7**, we have $(\sigma', O') \models \psi$.

Mutation, "$v.a\ :=\ x$": Suppose (σ, O) satisfy the precondition, i.e., $(\sigma, O) \models (\exists r_1, r_2 \cdot (v = r_1) \wedge (x = r_2) \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 -\!\!* \psi)))$, then we know that there exists r_1, r_2 such that $\sigma v = r_1$ and $\sigma x = r_2$. From the precondition we have $(\sigma v, a) \in \mathsf{dom}_2\, O$, otherwise $r_1.a \mapsto -$ will be $\texttt{false}$. So the command does not get stuck. Then by the operational semantics we have $(\sigma', O') = (\sigma, O \oplus \{(r_1, a) \mapsto r_2\})$. By the definitions of $*$ and $-\!\!*$, we can deduce that $(\sigma', O') \models \psi$.

Lookup, "$x\ :=\ v.a$": Suppose (σ, O) satisfy the precondition, that is, $(\sigma, O) \models (\exists r_1, r_2 \cdot (v = r_1) \wedge (r_1.a \hookrightarrow r_2) \wedge \psi[r_2/x])$. Then we know that that there exists r_1, r_2 satisfying $\sigma v = r_1$ and $O(r_1, a) = r_2$. From the precondition we have $(\sigma v, a) \in \mathsf{dom}_2\, O$, otherwise $r_1.a \hookrightarrow r_2$ will be $\texttt{false}$. So the command does not get stuck. Then by the operational semantics we have $(\sigma', O') = (\sigma \oplus \{x \mapsto r_2\}, O)$. Then by **Lemma 7**, we have $(\sigma', O') \models \psi$.

Cast, "$x\ :=\ (N)v$": Suppose $(\sigma, O) \models \exists r \cdot \mathsf{otype}(r) <: N \wedge v = r \wedge \psi[v/x]$. From $\exists r \cdot \mathsf{otype}(r) <: N \wedge v = r$, we have $\mathsf{otype}(\sigma v) <: N$, so by the operational semantic, we have $(\sigma', O') = (\sigma \oplus \{x \mapsto \sigma v\}, O)$. By **Lemma 7**, the conclusion holds.

Return, "$\texttt{return}\ \ e$": The proof is similar to the plain assignment.

Method Invocation, "$x\ :=\ v.m(\overline{e})$": Suppose $[\![T.m]\!] = F$ and $(\sigma, O) \models \exists r \cdot v = r \wedge F(v, \overline{e})(\psi[\mathsf{res}/x]))$. Since the command is well-typed, there exists $r : S$, $S <: T$ and $v = r$. By operational semantics, method m in class S is invoked. Suppose $[\![S.m]\!] = G$ and

$$\langle c, (\{\texttt{this} \mapsto \sigma v, \overline{z \mapsto \sigma e}, \overline{y \mapsto \mathsf{nil}}, \mathsf{res} \mapsto \mathsf{nil}\}, O)\rangle \leadsto^* (\sigma', O'),$$

where c is the body command of method m in S, we need to show:

$$(\sigma, O) \models F(v, \overline{e})(\psi[\mathsf{res}/x])) \quad \text{implies} \quad (\sigma \oplus \{x \mapsto \sigma' \mathsf{res}\}, O') \models \psi.$$

Assume σ does not contain variables $\overline{z}$ and $\overline{y}$. This can be achieved by renaming local variables involved in c, and renaming $\texttt{this}$ as $\texttt{this}_0$, res as res_0. Let

$$\sigma_0 = \{\texttt{this}_0 \mapsto \sigma v, \overline{z \mapsto \sigma e}, \overline{y \mapsto \mathsf{nil}}, \mathsf{res}_0 \mapsto \mathsf{nil}\},$$

then $\mathsf{dom}\, \sigma_0 \cap \mathsf{dom}\, \sigma = \emptyset$, and σ_0 does not contain variables in $F(v, \overline{e})(\psi[\mathsf{res}/x])$. By $(\sigma, O) \models F(v, \overline{e})(\psi[\mathsf{res}/x])$ and **Lemma 6**, we have:

$$(\sigma \cup \sigma_0, O) \models F(v, \overline{e})(\psi[\mathsf{res}/x]).$$

Because $F = \lambda \texttt{this}, \overline{z} \cdot \lambda \psi \cdot f(\psi)[\overline{\mathsf{nil}}/\overline{y}]$, where $F(v, \overline{e})(\psi[\mathsf{res}/x])$ can be viewed as an assertion that replace $\texttt{this}$, $\overline{z}$ and $\overline{y}$ in $f(\psi)$ to v, $\overline{e}$ and nil, and we already have $\sigma_0 \texttt{this} = \sigma v$, $\overline{\sigma_0 z = \sigma e}$, and $\overline{\sigma_0 y} = \mathsf{nil}$, so by **Lemma 7** we have:

$$(\sigma \cup \sigma_0, O) \models f(\psi[\mathsf{res}/x]).$$

By inductive hypothesis we have $\langle c, (\sigma_0, O)\rangle \rightsquigarrow^* (\sigma_0', O')$. Because c does not contain variables in σ, then by **Lemma 14**, we have $\langle c, (\sigma \cup \sigma_0, O)\rangle \rightsquigarrow^* (\sigma \cup \sigma_0', O')$. Since $[\![\Gamma, S, m \vdash c : \mathbf{com}]\!] = f$, by induction hypothesis, $(\sigma \cup \sigma_0', O') \models \psi[res/x]$, where res is the return value. Note that by the operational semantics, the return value is stored in res_0, then

$$(\sigma \oplus \{x \mapsto \sigma_0'\mathsf{res}_0\} \cup \sigma_0', O') \models \psi.$$

Because ψ does not contain variables in σ_0', by **Lemma 6**,

$$(\sigma \oplus \{x \mapsto \sigma_0'\mathsf{res}_0\}, O') \models \psi.$$

Now we rename variables in σ_0' back and have $(\sigma \oplus \{x \mapsto \sigma'\mathsf{res}\}, O') \models \psi$. The conclusion holds.

Object Creation: Suppose $(\sigma, O) \models \forall r \cdot \mathsf{raw}(r, N) \mathrel{-\!\!*} F(r, \overline{e})(\psi[r/x])$, then for any $r \notin O$,

$$(\sigma, O \oplus \{r \mapsto \{\overline{a \mapsto \mathsf{nil}}\}\}) \models F(r, \overline{e})(\psi[r/x]).$$

Similar to Method Invocation, assume σ does not contain variables $\overline{z}$ and $\overline{y}$, and let

$$\sigma_0 = \{\mathtt{this}_0 \mapsto r, \overline{z \mapsto \sigma e}, \overline{y \mapsto \mathsf{nil}}\},$$

then $\mathsf{dom}\,\sigma_0 \cap \mathsf{dom}\,\sigma = \emptyset$, and σ_0 does not contain variables in $F(v, \overline{e})(\psi[r/x])$. By **Lemma 6**, we have:

$$(\sigma \cup \sigma_0, O \oplus \{r \mapsto \{\overline{a \mapsto \mathsf{nil}}\}\}) \models F(v, \overline{e})(\psi[r/x]),$$

Unfolding F and noting that $\sigma_0\mathtt{this} = r$, $\overline{\sigma_0 z = \sigma e}$, and $\overline{\sigma_0 y = \mathsf{nil}}$, we have

$$(\sigma \cup \sigma_0, O \oplus \{r \mapsto \{\overline{a \mapsto \mathsf{nil}}\}\}) \models f(\psi[r/x]),$$

By operational semantics and induction hypothesis, we have

$$(\sigma \cup \sigma_0', O') \models \psi[r/x],$$

where σ_0 and O' is defined by the operational semantics rule (O-NEW). Then because $\psi[r/x]$ does not contains variables in σ_0, by **Lemma 6**, we have $(\sigma, O') \models \psi[r/x]$, then we have $(\sigma \oplus \{x \mapsto r\}, O') \models \psi$. The conclusion holds.

Until now we have finished the proof for the soundness. □

Appendix B.3. Completeness

Now we prove the completeness theorem of the WP semantics (**Theorem 5**).

Proof. We prove that, for any $\psi, \psi' \in \Psi$ and $c \in \mathbf{COM}$ satisfying $[\![\Gamma, C \vdash c : \mathbf{com}]\!]\psi = \psi'$, then for any pair of states (σ, O) and (σ', O'), if $(\sigma', O') \models \psi$ and $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, then $(\sigma, O) \models \psi'$. We prove it by induction on the structure of commands. We adopt some notations used in defining the WP semantics, and always assume ψ the postcondition.

Sequential Composition, "c_1; c_2": Suppose $\langle c_1, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, and $\langle c_2, (\sigma', O')\rangle \rightsquigarrow^* (\sigma'', O'')$, and $(\sigma'', O'') \models \psi$. Then by induction hypothesis, we have $(\sigma', O') \models f_2(\psi)$ and $(\sigma, O) \models f_1(f_2(\psi))$. So the conclusion holds.

Condition, "`if` $b\ c_1$ `else` c_2": Assume $\llbracket b \rrbracket_{(\sigma,O)} = \mathsf{rtrue}$, and suppose $\langle c_1, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$ and $(\sigma', O') \models \psi$. By induction hypothesis we have $(\sigma, O) \models f_1(\psi)$, then $(\sigma, O) \models (b \Rightarrow f_1(\psi)) \wedge (\neg b \Rightarrow f_2(\psi))$, that is $(\sigma, O) \models \psi'$. The case when $\llbracket b \rrbracket_{(\sigma,O)} = \mathsf{rfalse}$ can be proved similarly, so, the conclusion holds.

Iteration, "`while` $b\ c$": We treat `while` $b\ c$ as `if` $b\ c$; `while` $b\ c$ `else skip`, by the definition of G and induction hypothesis, the conclusion holds.

Skip: The proof is trivial.

Assignment, "$x := e$": Suppose $\langle x := e, (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto \sigma e\}, O)$, and $(\sigma \oplus \{x \mapsto \sigma e\}, O) \models \psi$. By **Lemma 7**, $(\sigma, O) \models \psi[e/x]$. The conclusion holds.

Mutation, "$v.a := x$": Suppose $\langle v.a := x, (\sigma, O)\rangle \rightsquigarrow (\sigma, O \oplus \{(\sigma v, a) \mapsto \sigma x\})$, and $(\sigma, O') \models \psi$, where $O' = O \oplus \{(\sigma v, a) \mapsto \sigma x\}\}$. We need to show $(\sigma, O) \models \psi'$, where

$$\psi' = \exists r_1, r_2 \cdot (v = r_1) \wedge (x = r_2) \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 \mathrel{-\!\!*} \psi))$$

By the premise of Mutation, let $r_1 = \sigma v, r_2 = \sigma x$, then we have

$$(\sigma, O) \models \exists r_1, r_2 \cdot (v = r_1) \wedge (x = r_2) \quad \text{and} \quad O' = O \oplus \{(r_1, a) \mapsto r_2\}.$$

Since $(\sigma, O') \models \psi$, we have

$$(\sigma, O' - \{(r_1, a) \mapsto r_2\}) \models (r_1.a \mapsto r_2) \mathrel{-\!\!*} \psi.$$

Then by the definition of $*$ and $\mathrel{-\!\!*}$, we have

$$(\sigma, O) \models r_1.a \mapsto - * (r_1.a \mapsto r_2 \mathrel{-\!\!*} \psi).$$

So the conclusion holds.

Lookup, "$x := v.a$" : Suppose $\langle x := v.a, (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto O(\sigma v)(a)\}, O)$, and $(\sigma \oplus \{x \mapsto O(\sigma v)(a)\}, O) \models \psi$. Then we know there exists r_1, r_2 that $\sigma v = r_1, O(r_1, a) = r_2$, so

$$(\sigma \oplus \{x \mapsto r_2\}, O) \models \exists r_1, r_2 \cdot (v = r_1) \wedge (r_1.a \hookrightarrow r_2).$$

Then, by **Lemma 7**, we have

$$(\sigma, O) \models \exists r_1, r_2 \cdot (v = r_1) \wedge (r_1.a \hookrightarrow r_2) \wedge \psi[r_2/x].$$

So the conclusion holds.

Cast, "$x := (N)v$": Similar to Assignment, just pay attention that when the command executes successfully and $\mathsf{otype}(\sigma v) <: N$. These imply $(\sigma, O) \models \exists r \cdot \mathsf{otype}(r) <: N \wedge v = r$. So the conclusion holds.

Return, "`return` e": Suppose $\langle \texttt{return}\ e, (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{\mathsf{res} \mapsto \sigma e\}, O)$, and $(\sigma \oplus \{\mathsf{res} \mapsto \sigma e\}, O) \models \psi$. Then $(\sigma, O) \models \psi[\mathsf{res}/e]$. The conclusion holds.

Method Invocation, "$x := v.m(\overline{e})$": Suppose

$$\langle x := v.m(\overline{e}), (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto \sigma'\mathsf{res}\}, O') \quad \text{and} \quad (\sigma \oplus \{x \mapsto \sigma'\mathsf{res}\}, O') \models \psi,$$

where σ' and O' are defined by the operational semantics. By the type system and operational semantics, the value of v must be a reference r and i with $\mathsf{otype}(r) = S_i <: T$, and S_i contains a definition for method m. So, we suppose c_i the body command of method m in class S_i, and that $[\![\Gamma, S_i, m \vdash c : \mathbf{com}]\!] = f_i$.

Now, similar to the proof of **Theorem 4** (the Soundness Theorem), we rename variables in σ' such that $\mathsf{dom}\,\sigma' \cap \mathsf{dom}\,\sigma = \emptyset$, and let

$$\sigma_0 = \{\mathtt{this}_0 \mapsto \sigma v, \overline{z \mapsto \sigma e}, \overline{y \mapsto \mathsf{nil}}, \mathsf{res}_0 \mapsto \mathsf{nil}\}.$$

Here σ_0 does not contain variables in σ, nor in ψ, and let σ_0' be a store satisfying

$$\langle c_i, (\sigma_0, O)\rangle \rightsquigarrow^* (\sigma_0', O').$$

By hypothesis for Assignment and **Lemma 6** we have $(\sigma \cup \sigma_0', O') \models \psi[\mathsf{res}_0/x]$. By **Lemma 14** we also have $\langle c_i, (\sigma \cup \sigma_0, O)\rangle \rightsquigarrow^* (\sigma \cup \sigma_0', O')$. Then by induction hypothesis we have $(\sigma \cup \sigma_0, O) \models f_i(\psi[\mathsf{res}_0/x])$. Since $\sigma_0\mathtt{this}_0 = \sigma v$, $\overline{\sigma_0 z = e}$, and $\overline{\sigma_0 y = \mathsf{nil}}$, we can do substitution that:

$$(\sigma \cup \sigma_0, O) \models f_i(\psi[\mathsf{res}_0/x])[v, \overline{e}, \overline{\mathsf{nil}}/\mathtt{this}_0, \overline{z}, \overline{y}],$$

This is equivalent to

$$(\sigma \cup \sigma_0, O) \models F_i(v, \overline{e})(\psi[\mathsf{res}_0/x]),$$

Because $F_i(\psi)$ does not contains variable in σ_0, so by **Lemma 6**, and by substituting x with res_0, then we get:

$$(\sigma, O) \models F_i(v, \overline{e})(\psi[\mathsf{res}_0/x]),$$

Recall that we just need one particular name res to denote the return value of a method, we can obtain:

$$(\sigma, O) \models F_i(v, \overline{e})(\psi[\mathsf{res}/x]),$$

According to the proof above, we have:

$$(\sigma, O) \models \exists r \neq \mathsf{rnull} \cdot v = r \wedge (r : S_i \wedge F_i(v, \overline{e})(\psi[\mathsf{res}/x]))$$

At last, by the property of $\bigvee$, we have

$$(\sigma, O) \models \exists r \neq \mathsf{rnull} \cdot v = r \wedge (\bigvee (r : S_i \wedge F_i(v, \overline{e})(\psi[\mathsf{res}/x])))$$

The conclusion holds.

Object Creation, "$x := \mathtt{new}\, N(\overline{e})$": Suppose

$$\langle x := \mathtt{new}\, N(\overline{e}), (\sigma, O)\rangle \rightsquigarrow^* (\sigma \oplus \{x \mapsto r\}, O') \quad \text{and} \quad (\sigma \oplus \{x \mapsto r\}, O') \models \psi,$$

where O' is defined by the operational semantics. Similar to Method Invocation, let

$$\sigma_0 = \{\mathtt{this}_0 \mapsto r, \overline{z \mapsto \sigma e}, \overline{y \mapsto \mathsf{nil}}\},$$

Here we can assume that σ_0 does not contain variables in σ and ψ. Let σ'_0 be the store satisfying $\langle c, (\sigma_0, O)\rangle \rightsquigarrow^* (\sigma'_0, O')$, where c is the body of N's constructor. By hypothesis for Assignment and **Lemma 6** we have $(\sigma \cup \sigma'_0, O') \models \psi[r/x]$. Then by induction hypothesis

$$(\sigma \cup \sigma_0, O \oplus \{r \mapsto \{\overline{a \mapsto \mathsf{nil}}\}\}) \models f(\psi[r/x]).$$

Since $\sigma_0\mathtt{this} = r, \overline{\sigma z = e}, \overline{\sigma y = \mathsf{nil}}$, we can do substitution that:

$$(\sigma \cup \sigma_0, O \oplus \{r \mapsto \{\overline{a \mapsto \mathsf{nil}}\}\}) \models f(\psi[r/x])[r, \overline{e}, \overline{\mathsf{nil}}/\mathtt{this}, \overline{z}, \overline{y}],$$

This is in fact $(\sigma \cup \sigma_0, O \oplus \{r \mapsto \{\overline{a \mapsto \mathsf{nil}}\}\}) \models F(r, \overline{e})(\psi[r/x])$. And now, $F(r, \overline{e})(\psi[r/x])$ does not contains variables in σ_0, so by **Lemma 6** we have:

$$(\sigma, O \oplus \{r \mapsto \{\overline{a \mapsto \mathsf{nil}}\}\}) \models F(r, \overline{e})(\psi[r/x]),$$

Then by the properties of $-\!\!*$,

$$(\sigma, O) \models \mathsf{raw}(r, N) -\!\!* F(r, \overline{e})(\psi[r/x]).$$

then by the operational semantics, r can be any reference that $r \notin O$, so

$$(\sigma, O) \models \forall r \cdot \mathsf{raw}(r, N) -\!\!* F(r, \overline{e})(\psi[r/x]).$$

This concludes the proof for the completeness of the WP semantics. $\square$

The proofs of soundness and completeness justify the WP semantics defined in Section 3.2.

References

[1] Abrial, J.-R., 1996. The B-Book: Assigning Programs to Meaning. Cambridge Univerisity Press.

[2] Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., Schulte, W., Rustan, K., Leino, M., 2003. Verification of object-oriented programs with invariants. Journal of Object Technology 3, 2004.

[3] Barnett, M., Leino, K. R. M., 2005. Weakest-precondition of unstructured programs. In: PASTE '05. ACM, New York, NY, USA, pp. 82–87.
URL `http://doi.acm.org/10.1145/1108792.1108813`

[4] Barnett, M., Leino, K. R. M., Schulte, W., 2005. The Spec# programming system: An overview. In: CASSIS 2004. Vol. 3362 of LNCS. Springer, pp. 49–69.

[5] Borba, P., Sampaio, A., 2000. Basic laws of ROOL: an object-oriented language. RITA 7 (1), 49–68.

[6] Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M., 2004. Algebraic reasoning for object-oriented programming. Science of Computer Programming 52 (1-3), 53 – 100.

[7] Borba, P., Sampaio, A., Cornélio, M., 2003. A refinement algebra for object-oriented programming. In: ECOOP'03. Vol. 2743 of LNCS. Springer, pp. 457–482.

[8] Burdy, L., Requet, A., Lanet, J.-L., 2003. Java applet correctness: A developer-oriented approach. In: Araki, K., Gnesi, S., Mandrioli, D. (Eds.), FME 2003. Vol. 2805 of LNCS. Springer, pp. 422–439.

[9] Cavalcanti, A., Naumann, D., 1999. A weakest precondition semantics for an object-oriented language of refinement. In: FME'99. Vol. 1709 of LNCS. Springer, pp. 1439–1459.

[10] Cavalcanti, A., Naumann, D., 2000. A weakest precondition semantics for refinement of object-oriented programs. IEEE Trans. on Software Engineering 26 (8), 713–728.

[11] Cornélio, M., Cavalcanti, A., Sampaio, A., Nov. 2002. Refactoring by transformation. Electronic Notes in Theoretical Computer Science 70 (3), 311–330.
URL `http://dx.doi.org/10.1016/S1571-0661(05)82564-2`

[12] de Boer, F. S., 1999. A WP-calculus for OO. In: Thomas, W. (Ed.), Foundations of Software Science and Computation Structures. Vol. 1578 of LNCS. Springer, pp. 135–140.
URL `http://dx.doi.org/10.1007/3-540-49019-1_10`

[13] Dijkstra, E. W., August 1975. Guarded commands, nondeterminacy and formal derivation of program. Communications of the ACM 18 (8), 453C457.

[14] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., Stata, R., May 2002. Extended static checking for java. SIGPLAN Notices 37, 234–245.
URL `http://doi.acm.org/10.1145/543552.512558`

[15] Fowler, M., 2000. Refectoring: Improving the Design of Existing Code. Addison-Wesley.

[16] Hesselink, W. H., February 1989. Predicate-transformer semantics of general recursion. Acta Informatica 26, 309–332.
URL `http://portal.acm.org/citation.cfm?id=69304.69306`

[17] Hoare, C. A. R., 1972. Proof of correctness of data representations. Acta Informatica 1, 271–281.

[18] Jacobs, B., 2002. Weakest precondition reasoning for java programs with jml annotations. Journal of Logic and Algebraic Programming 58, 2004.

[19] Jifeng, H., Li, X., Liu, Z., November 2006. rCOS: a refinement calculus of object systems. Theoretical Computer Science 365, 109–142.
URL `http://portal.acm.org/citation.cfm?id=1226608.1226614`

[20] Leavens, G., Baker, A., Ruby, C., 2006. Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Software Engineering Notes 31 (3), 1–38.

[21] Leavens, G., Leino, K. R. M., Müller, P., 2007. Specification and verification challenges for sequential object-oriented programs. Formal Aspects on Computing 19 (2), 159–189.

[22] Leavens, G. T., Naumann, D. A., Dec. 2006. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Tech. Rep. 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[23] Leavens, G. T., Naumann, D. A., Sep. 2006. Behavioral subtyping, specification inheritance, and modular reasoning. Tech. Rep. 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[24] Leino, K. R. M., Müller, P., 2004. Object invariants in dynamic contexts. In: Odersky, M. (Ed.), ECOOP 2004. Vol. 3086 of LNCS. Springer, pp. 95–108.

[25] Liskov, B., 1987. Keynote address - data abstraction and hierarchy. In: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum). OOPSLA '87. ACM, New York, NY, USA, pp. 17–34.
URL `http://doi.acm.org/10.1145/62138.62141`

[26] Liskov, B., Wing, J. M., 1994. A behavioral notion of subtyping. ACM Transactions on Programing Languages and Systems 16 (6), 1811–1841.

[27] Liu, Y., Hong, A., Qiu, Z., 2011. Inheritance and modularity in specification and verification of oo programs. In: TASE 2011. IEEE Computer Society, Xi'an, China, pp. 19–26.

[28] Liu, Y., Qiu, Z., 2010. A separation logic for OO programs. Tech. Rep. 2010-42, School of Math., Peking University, avaliable at http://www.mathinst.pku.edu.cn/index.php?styleid=2, Preprints.

[29] Liu, Y., Qiu, Z., 2011. A separation logic for OO programs. In: FACS 2010. Vol. 6921 of LNCS. Springer, Guimaraes, Portugal.

[30] Liu, Y., Qiu, Z., Long, Q., 2010. A weakest precondition semantics for Java. Tech. Rep. 2010-46, School of Math., Peking University, avaliable at http://www.mathinst.pku.edu.cn/index.php?styleid=2, Preprints.

[31] Liu, Y., Qiu, Z., Long, Q., 2011. WP semantics and behavioral subtyping. In: ICTAC 2011. Vol. 6916 of LNCS. Springer, Johannesburg, South Africa, pp. 154–172.

[32] Morgan, C., 1998. Programming from Specifications. Prentice/Hall.

[33] Müller, P., 2000. Modular Specification and Verification of Object-Oriented Programs. Vol. 2262 of LNCS. Springer.

[34] Noble, J., Vitek, J., Potter, J., 1998. Flexible alias protection. In: Jul, E. (Ed.), ECOOP'98. Vol. 1445 of LNCS. Springer, pp. 158–185.

[35] Parkinson, M., 2005. Local reasoning for Java. Ph.D. thesis, University of Cambridge.

[36] Parkinson, M., Summers, A., 2011. The relationship between separation logic and implicit dynamic frames. In: Barthe, G. (Ed.), Programming Languages and Systems. Vol. 6602 of LNCS. Springer, pp. 439–458.

[37] Parkinson, M. J., Bierman, G. M., 2008. Separation logic, abstraction and inheritance. In: POPL '08. ACM, New York, NY, USA, pp. 75–86.
URL `http://doi.acm.org/10.1145/1328438.1328451`

[38] Pierik, C., de Boer, F. S., 2003. A syntax-directed hoare logic for object-oriented programming concepts. In: Najm, E., Nestmann, U., Stevens, P. (Eds.), FMOODS 2003. Vol. 2884 of LNCS. Springer Berlin / Heidelberg, pp. 64–78.

[39] Pierik, C., de Boer, F. S., 2005. A proof outline logic for object-oriented programming. Theoretical Computer Science 343 (3), 413–442.

[40] Qiu, Z., Wang, S., Long, Q., 2007. Sequential μJava: Formal foundations. Tech. Rep. 2007-35, School of Math., Peking University, avaliable at http://www.mathinst.pku.edu.cn/index.php?styleid=2, Preprints.

[41] Reynolds, J. C., 2002. Separation Logic: A logic for shared mutable data structures. In: LICS 2002. IEEE Computer

Society, Los Alamitos, CA, USA, pp. 55–74.

[42] Smans, J., Jacobs, B., Piessens, F., 2009. Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (Ed.), ECOOP 2009. Vol. 5653 of LNCS. Springer, pp. 148–172.

[43] Wang, S., Barbosa, L. S., Oliveira, J. N., 2008. A relational model for Confined Separation Logic. In: TASE 2008. IEEE Computer Society, Nanjing, China, pp. 263–270.