Weakest Precondition Semantics for OO Programs: 
A Separation Logic Approach

Yijing Liu, Quan Long, and Qiu Zongyan

LMAM and Department of Informatics, School of Mathematical Sciences, Peking University
Email: {liuyj, longquan, qzy}@math.pku.edu.cn

Abstract. For the object oriented paradigm, providing a relatively rich model language equipped with formal semantics for practical reasoning is an important and long-standing open problem. In this work, \textit{\mu}Java, a sufficient large subset of sequential Java is defined. An OO Separation Logic with pure reference semantic model is developed. Facilitated by this logic, the Weakest Precondition (WP) semantics for \textit{\mu}Java is defined, and its soundness and completeness are proved. As far as we know, this is the first work on the completeness of such a semantics. Some key properties are shown still hold, especially the frame rule that is important for local reasoning. Additionally, we find some properties absent in the original Separation Logic, but important for OO reasoning. We introduce Hoare Triple based on the WP semantics. As the application and illustration of how the WP semantics serve the verification of OO programs, some examples are given, with the class invariant proof in a case study. We anticipate that this work would be helpful for the disciplines of OO software verification and refinement.

Keywords: Object Orientation, Weakest Precondition, Separation Logic, Verification

1 Introduction

For both software development and programming languages design, it is evident that object-orientation (OO) is and will remain important, because it supports many extremely useful abstractions. It is well-known that OO features, say, inheritance, object references, and dynamic binding bring great benefits to the development, reuse and maintenance of software. However, for program verification, challenges arisen over the OO paradigm are still long standing open problems. There are two key issues mutually depending on each other: (1) building proper formal models for OO languages, and (2) developing useful methods for specifying and verifying OO programs.

For the first issue, researchers have proposed many formal frameworks to describe core concepts of OO, which are too many to be enumerated. Due to the complicated state space of OO programs, different models are used in different frameworks. Roughly, the notable models beneath the frameworks can be classified as \textit{Object Graph Model}, \textit{Access Trace Model}, and \textit{Stack Heap Model}.

The Object Graph Models treat the program state as a graph, such as the topological mode in [21], or the object diagram in [8, 30]. The whole graph gives a state snapshot in the execution, where vertexes denote objects, and edges denote variables and object
attributes. Models of this kind are intuitive and always independent of languages. But it is not easy to define a proper and useful semantics based on them, since the abstract and useful operations on the graphs are hard to be defined adequately.

The Trace Model is first introduced by [14], where each object is identified by a set of traces that are paths to the object. The models excel in alias analysis, as shown by [5]. But for general purpose analysis, these models do not behave well, since the concept of traces is too abstract that it is hard to define easy-use semantics based on a model of this kind. One attempt on this direction is [8].

The Stack Heap Model is an extension of classical store model, with an additional heap (a map from address to values), to represent program states. These models seem low-level, however, various forms of semantics are relatively easy to be defined based on them. Some semantic works have been done here, e.g. [24].

Although many interesting work have been done, each of them dealt with some OO features remarkably, a full accounting of all important OO features is still lack.

For the second issue, many jobs have been done to specify the behavior of OO programs. Although the work related to JML [17] and Spec# [2] have got increasing attention, many central issues of OO, especially which related to the mutable object structures, have not been considered there. A great deal of semantic issues must be understood for a big-leap in this field. In this article, we utilize the Separation Logic and Weakest Precondition (WP) semantics to address various issues deeply relative to formal model and verification of OO programs.

Separation Logic [28, 22] is a powerful tool to handle shared mutable data structure. Based on it, people have developed many techniques to verify C-like programs, and sometimes aim to OO programs. However, it is not straightforward to use original Separation Logic to specify and verify OO programs, because the underlying storage model of Separation Logic is not ready for many OO concepts. For example, there is no correspondents of object attributes in the model. Although some works directly use Separation Logic to specify OO programs, such as [9] etc., they often involve too much implementation details, and do not supply a general model for OO. Parkinson et al did some work to modify Separation Logic for OO, e.g. [24, 25], where a revised model is introduced to represent program states, and an Separation Logic with intuitionistic semantics is developed for reasoning.

WP semantics is a powerful tool for program verification, refinement, etc. According to our knowledge, people make efforts on a WP semantics for OO since 1999. In [11], a WP calculus is proposed for OO programs toward to support some object sharing. Besides many restrictions to the programs and assertions, the main problem is that the formalism for expressing the semantics is restricted to the form of syntactic substitution. For example, the substitution for assignments is defined as:

\[
 l.x[e/x] = \text{if } (l[e/x]) = \text{self} \text{ then } e \text{ else } (l[e/x]).x \text{ fi}
\]

where \( l = \text{self} \) stands for that \( l \) refers to current object. With this technique, paid many prices of discussing special cases\(^1\), the semantics is defined syntactically. However, \( l =

\(^1\) We found an example permitted by the restrictions but failed to be included in the definition. Although it is easy to remedy, is shows that managing the semantics in such a way is troublesome.
self cannot be checked syntactically. This makes the effort of defining WP semantics syntactically questionable. The similar problem appeared in the later work [26] which attempts to give a pure syntax-based Hoare Logic for OO programs.

A. Cavalcanti and D. Naumann made a significant contribution to give a WP semantics for OO languages. In [6, 7] they studied a language with subtyping, dynamic binding, but not sharing. Supported by a typing environment, each command is associated with a semantics as a predicate transformer. The notion of OO refinement is defined too. Based on their former work, [10] presented a set of refactoring laws taken from [13]. Further, [4] proposed a refinement algebra for OO programming. An extended paper [3] summarized the work. But, the semantics models in these work (e.g., in [6, 7]) were not based on references, this might be departure from an essential feature of OO languages, and make the object sharing and updating hard to treat, if not impossible. A noticeable example appear in [7], where only WP semantics of assignment with upcast can be defined. It is due to, as far as our opinion, the neglecting of object sharing. In [4] and [10], no refinement law related to references or pointers is given. Further, in [10], there are a number of mistakes when some attempts were made to encode the sharing-related refactoring in non-reference semantic model. Thus, in summary, using a non-reference semantic model, it is problematic to verify many of OO concerns interested by industry.

Accounting to the procedural paradigm, WP semantics plays the central role in the semantics study, and the foundation stone for many theories deeply related to the software engineering, including specification, verification, refinement, programming from specifications [19], specification-based code generation, etc. A well-defined WP semantics would play the similar role in the OO paradigm. However, as discussed above, such a WP semantics is still not emerge yet. We would like to have a try, and try to remedy the problems mentioned above in our work.

In this article, we develop a WP semantics for a model language \(\mu\text{Java}\), which is a sufficient large subset of sequential Java, and covers most important OO features including reference types, subtypes, inheritance, dynamic binding, and sharing based parameters for methods. We use a revised Stack Heap Model as the underlying storage model of program states, and adopt a pure reference semantics, where every variable (as well as every attribute of an object) holds a reference. The Object Pool is defined to replace the heap in Separation Logic. Different to the heap, an Object Pool is a set of triple, that every triple \((r, a, r')\) means that reference \(r\) refers to some objects \(o\), and the value of attribute \(a\) of \(o\) is another reference \(r'\) that referring to another object \(o'\). With this model, we can conveniently define the operational semantics of \(\mu\text{Java}\).

Based on the storage model, we develop a revised Separation Logic, OO Separation Logic, which support all common OO features. The logic adopts classical semantics, which is more expressive than intuitionistic semantics by [16]. The logic can easily and precisely describe the program state, and form a promising approach to specify and reason features of main-stream OO programs. Properties of OO Separation Logic are investigated, especially the concept of separated assertions, which is extremely useful in reasoning OO programs.

With this logic, we develop a WP semantics for \(\mu\text{Java}\), and prove that it is sound and complete w.r.t the operational semantics. Then, we introduce Hoare Triple and corresponding WP based verification for OO programs. As a key step to show the system is
powerful and useful, we prove the frame rule [28], which is extremely important to local reasoning. We present some examples to illustrate the specification and verification of OO programs, as well as an approach to verify programs with class invariants.

The rest of this article is organized as follows. We introduce $\mu$Java in Section 2, with its type system, storage model and operation semantics. The OO Separation Logic, is developed in Section 3. The weakest precondition semantics is defined in Section 4, and some properties of the semantics are provided and proved, including the frame rule. We prove the soundness and completeness theorems in Section 5. We show how the specification and verification of OO programs can be carried out based on the WP semantics in Section 6. Then Section 7 concludes. Due to the page limitation, we omit some details and examples, which could be found in our reports [18] and [27].

2 $\mu$Java

The language investigated in this paper, $\mu$Java is a sequential subset of Java. What considered mainly are the essential OO features relating to object sharing, updating, and creation.

2.1 Syntax

The syntax for expressions and commands is as follows:

\[
\begin{align*}
  v & ::= \text{this} | x \\
  e & ::= \text{true} | \text{false} | \text{null} | v \\
  b & ::= \text{true} | \text{false} | e = e | \neg b | b \land b | b \lor b \\
  c & ::= \text{skip} | x := e | v.a := e | x := v.a | x := (C)v | x := v.m(\tau) \\
  & \quad x := \text{new } C(\tau) | \text{return } e | c; c | \text{if } b \text{ then } c \text{ else } c | \text{while } b \text{ c}
\end{align*}
\]

where $x$ denotes a variable, $C$ a class, $T$ a type, while $a$ and $m$ attribute and method respectively.

Here are some explanations:

- Similar to [1], to keep the semantics simple and clear, we have boolean the only primitive type, while $\text{true}$ and $\text{false}$ the only primitive values. Other primitive types can be added without substantial difficulties.
- We adopt the restricted forms for expressions whose values are independent of the Object Pool. The more complex expressions can be encoded with the help of assignments and auxiliary variables.
- The assignments are also restricted to a number of special forms. And other general cases can be also encoded by these forms. For instance, one can use $x := y.a$ and then refer to $x.a'$ as a replacement of $y.a.a'$.
- We consider “cast” as a part of commands rather than expressions. This is also a non-essential restriction.
- Command $x := \text{new } C(\tau)$ creates a new object, builds it with parameter $\tau$ and then assigns its reference to variable $x$. 
Similar to Java, we consider a method \textit{main} (if existing) a special one where the program starts. The syntax for classes and programs is as follows.

\[
T ::= \text{Bool} \mid \text{Object} \mid C \\
m\text{d} ::= T \text{m}(\overline{T} \overline{z})\{\overline{T} \overline{y}; \; c\} \\
c\text{d} ::= \text{class} \; C : C\{\overline{T} \overline{a}; C(\overline{T} \overline{z})\{\overline{T} \overline{y}; \; c\}; \overline{m}\text{d}\} \\
c\text{ds} ::= c\text{d} \mid c\text{d}; c\text{ds} \\
\text{prog} ::= c\text{ds}
\]

where \(m\) denotes a method, \(C\) a class. We do not have complex access control in \(\mu\text{Java}\) because it is not essential. A program is a sequence of class declarations.

Here are some explanations about the definition:

- We assume a build-in class \text{Object} as the super class of all user-defined classes, thus, each user-defined class has a direct super class. The (only) primitive type \text{Bool} is not a supertype or subtype of any other type. We assume an internal type \text{Null} as the type of \text{null}, which is the subtype of any classes. \text{Null} is not a feature of the language, thus cannot be used in programs. It is used only in the definitions for the type system and semantics.
- We use \(\overline{z}, \overline{y}\) to denote the parameters and local variables of methods, respectively, when the details are not important, and assume that \text{return} \(e\) is only appeared as the last statement of a non-constructor method. A special method \(C(\overline{T} \overline{z})\{\overline{T} \overline{y}; \; c\}\) in each class \(C\) services as the constructor, which must have the same name with the class.
- We assume all references to attributes in methods are decorated with \text{this}, to make the attribute references uniformly with the form \(v.a\). We can get rid of this restriction without any problem, but thus, some more rules should be added.
- We do not permit redefinition of attributes with the same name in subclasses, but permit method overriding.

In the design of \(\mu\text{Java}\), we put following points in mind:

- It should by be based on the commands and procedural, but not functional thus has states to make it model directly the main-stream OO languages.
- It should be as simple as possible to facilitate the theoretical use, but also large enough for covering the most important OO features in common practical languages. Thus we get rid of many non-important language details.
- We should choose the reference model for the variables and attributes of the objects, to reflect directly the reality in OO languages, and make many problems related to them on the surface, e.g., dynamic binding, object sharing, aliasing, casting, etc.
- We hope to have a clear separation of store operations and heap operations, for making the semantics relatively simple, tractable and understandable.

\subsection*{2.2 Static Environments and Typing}

In paper [7], a static environment was developed to localize and simplify the definition of the WP semantics. What we do here is similar. Our static environment consists of
two components: the typing environment $\Gamma$, and the method lookup environment $\Theta$. The environments are established by scanning the programs before execution. Because the technique is standard, we will not present the definitions and deduction rules here. Please refer our report [27] for the details. We are going to introduce only their skeletons w.r.t. the use in this paper.

Typing environment $\Gamma_{cds}$ records the static structural information of class declarations $cds$ with which the types of expressions or predicates can be derived, and the well-formedness of commands and methods can be checked. We abbreviate it as $\Gamma$ when there is no confusion. Formally, $\Gamma_{cds}$ is a tuple of the form:

\[(\text{cnames}, \text{methods}, \text{attrs}, \text{locvars})\]

where all the elements are relations over the classes, methods, and attributes, except for cnames which is a simple set:

- cnames: the set of all class names appearing in class declarations $cds$, with the predefined types $\text{Object}$, $\text{Bool}$ and $\text{Null}$.
- super: a map between class names, where $\text{super}(C_1, C_2)$ means $C_2$ is the immediate superclass of $C_1$. We define relation $\Gamma \vdash T_1 < : T_2$ as the transitive closure of super under $\Gamma$. Please note that, the only relation with boolean type is $\text{Bool} < : \text{Bool}$. We will often omit $\Gamma$ and write $T_1 < : T_2$ directly when it makes no confusion.
- methods: a relation from class names to method signatures. Here we use $\text{methods}(C, m(T_z : T))$ to stand for the fact that $m(T_z : T)$ is a method with this signature declared or inherited in $C$. We include also constructors in methods, which takes a special form $C(T_z)$ without the return type.
- attrs: a relation from classes to their attributes and the corresponding types, where $\text{attrs}(C, a : T)$ means attribute $a$ of type $T$ is defined in $C$. We will use $\Gamma.\text{attrs}(C)$ (or simply $\text{attrs}(C)$) to denote the attribute set of class $C$.
- locvars: a relation over classes, methods and variables, where $\text{locvars}(C, m, x : T)$ means that $x$ of type $T$ is a parameter or local variable in method $m$ of class $C$, thus is visible in the body of $m$.

The type of a local variable or parameter depends on the method scope. The type judgement takes the form $\Gamma, C, m \vdash e : T$ to denote that expression $e$ is of type $T$ in the scope of method $m$ of $C$ under typing environment $\Gamma$.

The main usage of the type system is to judge the well-formedness of commands. We use $\Gamma, C, m \vdash e : \text{com}$ to mean that $e$ is a well-formed command in body of $m$ of $C$ under $\Gamma$. The operational semantics and WP are defined only for well-formed commands. Again, due to the page limitation, we will omit the detailed definitions. Please refer to [27] if interested in. In this article, one can just imagine that the command $e$ is statically guaranteed correct by the type system when $\Gamma, C, m \vdash e : \text{com}$ holds. We consider only the well-formed programs in the rest part of the paper.

The other part of the static environment, $\Theta$, is used to look up method body. We will fix the method body code to each class statically, by mirroring the vtable technique used in OO practice, that is, each class copies all the inherited methods from its direct superclass (either the reference or the code).
$\Theta$ is composed of triples with the form $(C, m, m\text{th}(\overline{y}; c))$, where $C$ is a class name in $\text{cnames}$, $m$ is a method name of $C$, and $c$ is the body of $m$, which is a well-formed command. Because we deal with only the dynamic behavior of well-typed programs without the consideration of overriding methods, no type information is necessary in $\Theta$. For intuitive concern, we use $\Theta, C, m \rightarrow m\text{th}(\overline{y}; c)$ to represent the fact that $(C, m, m\text{th}(\overline{y}; c)) \in \Theta$.

For a method $m$ in class $C$ that $\Theta, C, m \rightarrow m\text{th}(\overline{y}; c)$. If $\Gamma, C, m \vdash c : \text{com}$, then $m$ is well-typed in $C$, denoted by “$\Gamma, C \vdash m : \text{method}$”.

In [27], we present the detailed rules for the construction of $\Gamma$ and $\Theta$ by scanning the program text, as well as for typing, which are all ignored here. And we have proven that an expression (or command) typing does determine a derivation.

**Theorem 1.** For a typing of the form $\Gamma, C, m \vdash e : T$ or $\Gamma, C, m \vdash b : \text{Bool}$, there is at most one derivation. For each typing $\Gamma, C, m \vdash c : \text{com}$, there is at most one derivation. □

### 2.3 The Storage Model

We define that the state of $\mu$Java programs consists of a store and an object pool, based on two basic sets:

- **Name**: an infinite set of names, used for representing variable names or attribute names in programs in the semantics.
- **Ref**: an infinite set of references that can be thought as addresses of objects. There is a special and distinguishable reference $\textit{rnull}$ which never refers to any object. Here we treat a reference as an atomic entity, where no operations are defined on them. This is different from the addresses in the Separation Logic where the address arithmetic is inherent and indispensable. Thus we can conclude that the storage model here is more abstract.

We define the sets of stores and object pools, respectively, as follows, as well as the set of states:

\[
\begin{align*}
\text{Store} & \equiv \text{Name} \rightarrow_{\text{fin}} \text{Ref} \\
\text{Opool} & \equiv \text{Ref} \rightarrow_{\text{fin}} \text{Name} \rightarrow_{\text{fin}} \text{Ref} \\
\text{State} & \equiv \text{Store} \times \text{Opool}
\end{align*}
\]

We will use meta-variables $\sigma$ and $O$, possibly with subscript, to denote elements of Store and Opool (object pools), respectively, thus a state is a pair ($\sigma, O$). A store $\sigma$ maps (variable) names to the references they record, and an Opool $O$ maps references to functions from (attribute) names to references. We assume three implicit constants references $\textit{rtrue}$, $\textit{rfalse}$ and $\textit{rnull}$ representing the values of $\text{true}$, $\text{false}$ and $\text{null}$, respectively, so that $\sigma\textit{true} = \textit{rtrue}$, $\sigma\textit{false} = \textit{rfalse}$ and $\sigma\textit{null} = \textit{rnull}$ for every $\sigma$.

An element of $O$ is a triple $(r, a, r')$, where $r$ is a reference to some object $o$, $a$ is the name of an attribute of $o$, and $r'$ is the reference recorded in attribute $a$ of $o$. We will always use $r$, possibly with subscripts, to denote a reference, i.e., $r, r_1, \ldots \in \text{Ref}$.

For the relation between $\text{Ref} \rightarrow (\text{Name} \rightarrow \text{Ref})$ and $\text{Ref} \times \text{Name} \rightarrow \text{Ref}$, when saying the domain of $O$, we might mean sometimes a subset of $\text{Ref}$ which is associated
with a set of objects, or a subset of \( \text{Ref} \times \text{Name} \) associated with a set of values (references). We will use \( \text{dom} \ O \) for the first case, and \( \text{dom} _2 O \) for the second case, thus \( \text{dom} _2 O \) is a set of pairs of the form \( \langle r, a \rangle \).

For any object in \( O \), we assume that its type (its class) can be obtained directly in run-time. For this becoming possible, we could have followed the common way in practice to include a special constant attribute in each object to record its type. Because this feature does not have critical effect on our discussion, we omit its details, and simply assume a function \( \text{type}(r) \) to get the type of the object referred by \( r \) at any time.

The store and Opool are modified by assignments. If current store is \( \sigma \), assignment \( x := y \) makes a new store \( \sigma' \) with all values of variables the same as \( \sigma \), except that \( \sigma' x = \sigma y \), formally, \( \sigma' = \sigma \oplus \{ x \mapsto \sigma y \} \). Assignments to attributes change Opool. If \( x \) refers to an object \( o \) in \( O \), i.e., \( O(\sigma x) = o \). Command \( x.a := y \) makes a new Opool \( O' \) where \( o(a) = \sigma y \), i.e., \( O'(\sigma ax)(a) = \sigma y \).

Most commands keep the domain of \( O \) unchanged, except object creation. If we create an object \( o \) and let \( x \) refer to it, the system will take a fresh reference \( r \), and let \( O' = O \oplus \{ r \mapsto o \} \) and \( \sigma' x = r \). We do not have release operation, and assume a garbage collector to take care the unreachable object. In [15], a semantics for an OO language that models garbage collection explicitly is given.

We borrow some notations from Separation Logic, use \( O_1 \perp O_2 \) to indicate that two Opoolels \( O_1 \) and \( O_2 \) have disjoint domains, i.e.,

\[
O_1 \perp O_2 \equiv \text{dom} _2 O_1 \cap \text{dom} _2 O_2 = \emptyset
\]

We use \( O_1 \ast O_2 \) to indicate the union of \( O_1 \) and \( O_2 \) when \( O_1 \perp O_2 \). Note that the basic cell in the model is an attribute of an object, and we separate cells in Opool.

### 2.4 Operational Semantics of \( \mu \text{Java} \)

As stated before, we take a pure reference semantics for \( \mu \text{Java} \), thus, a primitive type is also an object type. The values of \( e \) and \( b \) are determined by current state \( (\sigma, O) \).

We give the expression evaluation rules in Figure 1. Rules (BASIC-EXP) define the evaluation of basic expressions. Remember that there are only 4 different forms of basic expressions in \( \mu \text{Java} \). If \( v \notin \sigma \), the evaluation of \( v \) will go to abort. However, in a well-typed program, the evaluation of \( v \) always makes sense. Rules (BOOL-EXP) define the semantics of boolean expressions. Similar to \( \neg \), the semantics for operators \( \lor, \land, \text{etc.} \)
(c₁, (σ, O)) \leadsto^* (c', (σ', O')) \leadsto^* (c₁, (σ, O)) \leadsto^* \text{abort} \quad \text{(O-SEQ)}

σb = \text{true}, (c₁, (σ, O)) \leadsto^* \text{abort} \quad \text{(O-COND)}

(\text{if } b \text{ then } c \text{ else } c₂, (σ, O)) \leadsto^* \text{abort} \quad \text{(O-ITER)}

(x := e, (σ, O)) \leadsto (σ \oplus \{x \mapsto σe\}, O) \quad \text{(O-ASN-I)}

(v.a := e, (σ, O)) \leadsto (σ \oplus \{\{v.a \mapsto σe\}\}) \quad \text{(O-ASN-II)}

(x := v.a, (σ, O)) \leadsto (σ \oplus \{x \mapsto O(σv)(a)\}, O) \quad \text{(O-ASN-III)}

(x := (N)v, (σ, O)) \leadsto (σ \oplus \{x \mapsto σv\}, O) \quad \text{(O-ASN-C)}

\text{if} σv = C, \Theta, C, m = \text{mth}(τ)\{\text{var } τ; e\}

\langle c, (\text{this } → σv, x → σx, y → \text{nil, res } → \text{nil}, O) \rangle \leadsto^* (σ', O') \quad \text{(O-INV)}

\langle x := v.m(τ), (σ, O) \rangle \leadsto^* (σ \oplus \{x \mapsto σr\}, O') \quad \text{(O-NEW)}

\text{r is fresh}

\text{Fig. 2. Operational Semantics for } \mu\text{Java}

is standard, and is omitted here. Since the semantics of expressions are only dependent with the store, we will use σe and σb as the abbreviations for \|e\|_c(σ, O) and \|b\|_c(σ, O).

We assume every store contains an internal-variable res for recording the return value of recently invoked method, which is not allowed in programs. Because μJava is sequential, we just need one res to specify the return value for methods.

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 \text{abort} = (σ, O). The semantics is defined as a transition relation \leadsto^*: \text{Configuration} \leadsto^* \text{Configuration} \cup \{\text{abort}\}

Here Command is the set of valid program texts. A terminal configuration represents that the execution of a (piece of) program has completed successfully, while abort represents that the program goes wrong in execution, because of memory faults, wrong type casts, etc. We define \leadsto^* the finite transition closure of \leadsto.
Rules (O-SEQ), (O-COND), and (O-Iter) for structural commands are simple where \( \theta \) represents a terminal state or abort. An abort stops the execution, and reaches the abort directly.

The simple \( x := e \) is independent of Opool described by (O-ASN-I). Both the lookup and update operations look into the Opool. They go abort when dereferencing an attribute out of the Opool, that includes the cases where \( v \) has a null value. Rules (O-ASN-II) and (O-ASN-III) capture their behaviors.

Rule (O-ASN-C) shows that the 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.

Command \( x := \text{new } C(e) \) creates a new object of class \( C \), initiates its attributes (including inherited ones) with command \( c \), and lets \( x \) refer to the object. In (O-NEW), \( r \) is a fresh reference, whose selection is non-deterministic. Here \( \pi \) are the attribute sequence of class \( C \) including the inherited ones, and \( \{ r \mapsto \{ a \mapsto - \} \} \) stands for \( \{ (r, a, -) \mid a \in \text{attrs}(C) \} \), where "-" denotes any reference which we do not care.

Obviously, a program might fail to terminate for falling into an infinite iteration, or infinite recursive calls. In this case, the deduction cannot terminate too.

We have the following lemmas for store extension and shrink.

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

### 3 An OO Separation Logic

To facilitate OO features, almost all OO languages adopt the reference model, where both values of variables and attributes of objects are references to objects\(^2\). A special case is that the value can be null to mean referring to no object. This model induces a great possibility of sharing: besides different variables can share references, different attributes of objects can also share references, and can have sharing with variables. For modeling these features, we define an OO Separation Logic for OO specialities.

\(^2\) One exception might be variables and attributes of primitive types, while many languages use value model for them for efficiency.
\[ \sigma, O \models b \iff [b]_{(\sigma,O)} = \text{rtrue} \] (A-BOOL)

\[ \sigma, O \models e = r \iff [e]_{(\sigma,O)} = r \] (A-BASIC)

\[ \sigma, O \models \text{emp} \iff O = \emptyset \] (A-EMPTY)

\[ \sigma, O \models r_1.a \mapsto r_2 \iff O = \{(r_1, a, r_2)\} \] (A-SINGLE)

\[ \sigma, O \models \text{fresh}(r) \iff r \notin \text{dom} O \] (A-FRESH)

\[ \sigma, O \models r : T \iff r \in \text{dom} O \land \text{type}(r) = T \] (A-TYPE)

\[ \sigma, O \models p \land q \iff \exists O_1, O_2 \cdot O_1 \cdot O_2 = O \land \sigma, O_1 \models p \land \sigma, O_2 \models q \] (A-CONJ)

\[ \sigma, O \models p \rightarrow q \iff \forall O_1 \cdot O_1 \bot \sigma, O_1 \models p \Rightarrow \sigma, O_1 \cdot O \models q \] (A-IMPLY)

\[ \sigma, O \models \exists r \cdot \psi \iff \exists r \in \text{Ref} \cdot \sigma, O \models \psi \] (A-EX)

\[ \sigma, O \models \forall r \cdot \psi \iff \forall r \in \text{Ref} \cdot \sigma, O \models \psi \] (A-UNI)

Fig. 3. Semantics of Assertions

3.1 Assertions

The assertion language in the OO Separation Logic is similar to Separation Logic, with some revisions and extensions, to fit the special needs of OO programs. Its syntax is:

\[ \tau ::= S <: T \]

\[ \alpha ::= b \mid e = r \]

\[ \beta ::= \text{emp} \mid r_1.a \mapsto r_2 \]

\[ \gamma ::= r : T \mid r_1 = r_2 \mid \text{fresh}(r) \]

\[ \psi ::= \tau \mid \alpha \mid \beta \mid \gamma \mid \neg \psi \mid \psi \land \psi \mid \psi \lor \psi \mid \psi \Rightarrow \psi \mid \psi \land \psi \mid \psi \rightarrow \psi \mid \exists r \cdot \psi \mid \forall r \cdot \psi \]

where \( r \) denotes a reference, \( e \) and \( b \) are the expression and boolean expression respectively as defined in \( \mu \text{Java} \).

Basic assertions are divided into four categories, where

- \( \tau \) denotes assertions about types;
- \( \alpha \) denotes assertions involving only the store, and possibly references;
- \( \beta \) denotes assertions about empty and singleton Opoools, which take similar forms as in Separation Logic. Note that \( x.a \mapsto \ldots \) is not a valid assertion.
- \( \gamma \) talks about references, where \( r : T \) means \( r \) referring to an object of type \( T \). Here references are treated as atomic values. For any pair \( r_1 \) and \( r_2 \), \( r_1 = r_2 \) holds if \( r_1 \) and \( r_2 \) are the same. We treat \( r = e \) identical to \( e = r \). We add an assertion form \( \text{fresh}(r) \) to say that \( r \) is fresh with respect to the Opool.

We use \( \psi[e/x] \) (or \( \psi[r/x] \)) to denote the assertion built by substituting \( x \) with expression \( e \) (reference \( r \)) in \( \psi \). The substitution of references is not allowed.

The semantics of assertions is defined on state, where \( \sigma, O \models \psi \) means that \( \psi \) holds on state \( \sigma, O \). Clearly, the truth value of \( S <: T \) is determined by the program, thus can be determined statically. On the other hand, because references are atomic values, \( r_1 = r_2 \) is true only when \( r_1 \) and \( r_2 \) are the same reference. We will use \( \models S <: T \)
and $\models r_1 = r_2$ for them for convenience. The semantics of other assertions is defined in Fig. 3, except the definition for standard FOL combinators are omitted.

The boolean expressions get their values based on the state. The basic assertion asserts that the value of expression $e$ is $r$.

The assertion $\mathtt{emp}$ is true only when the current Opool is empty. The singleton assertion specifies a smallest non-empty Opool, similar to the singleton assertion in Separation Logic. Here an object field is taken as a cell. Please pay attention that the semantic of this assertion relies on the Opool only, without involving the variables and store. This is different from the Separation Logic.

An object has its type when created. As mentioned before, we assume that a function will hold on the extended Opool.

The boolean expressions get their values based on the state. The basic assertion specifies a smallest non-empty Opool, similar to the singleton assertion in Separation Logic. Here an object field is taken as a cell. Please pay attention that the semantic of this assertion relies on the Opool only, without involving the variables and store. This is different from the Separation Logic.

Besides the properties above, many propositions holding in Separation Logic hold also in OO Separation Logic. The rules (axiom schemata) for Separation Logic as
shown in the Section 3 of [28] are valid here. Some more rules can be proved (which are also valid in Separation Logic, ref. [29]), such as:

\[(p * q) \rightarrow r \iff p \Rightarrow (q \dashv r)\]

\[p * q \Leftrightarrow p * (p \dashv (p * q))\]

\[\text{emp} \Rightarrow p \dashv p\]

\[p \dashv q \Leftrightarrow p \dashv (p * (p \dashv q))\]

Intuitively, OO Separation Logic can be viewed as a domain-extended Separation Logic. That is, if we treat every tuple \((r, a)\) as an address of memory cell, and define a suitable address algorithm 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 involving in address arithmetic, will hold in OO Separation Logic. We will investigate the relation between Separation Logic and OO Separation Logic in future.

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

**Definition 1 (Separated Assertions).** Two assertions \(\psi\) and \(\psi'\) are separated of each other, iff for all stores \(\sigma\) and Opools \(O, O', \sigma, O \models \psi\) and \(\sigma, O' \models \psi'\) implies \(O \dashv O'\).

**Lemma 6.** \(r_1.a \dashv r_2.b\) are separated, provided that \(r_1 \neq r_2\), or \(a\) and \(b\) are different attribute names.

For example, suppose we have a Node class with two fields value and next, for a reference \(r : \text{Node}\), we know \(r.\text{value} \dashv -\) and \(r.\text{next} \dashv -\) are separated. No corresponding concept is in original Separation Logic, due to the absence of attributes.

**Lemma 7.** Suppose \(\psi_1\) and \(\psi_2\) are separated. If \(\sigma, O_1 \models \psi_1\) and \(\sigma, O_2 \models \psi_2\), then \(\sigma, O_1 \ast O_2 \models \psi_1 \ast \psi_2\).

**Lemma 8.** Suppose \(\psi_1\) and \(\psi_2\) are separated. If \(\sigma, O \models \psi_1 \ast \psi_2\), then exist an unique partition of \(O = O_1 \ast O_2\), such that \(\sigma, O_1 \models \psi_1\) and \(\sigma, O_2 \models \psi_2\).

**Lemma 9.** \(\psi_1\) is separated with \(\psi_2\) and \(\psi_3\), iff \(\psi_1\) is separated with \(\psi_2 \ast \psi_3\).

**Lemma 10.** Suppose \(\sigma, O \models \psi\) and \(\sigma, O \models \text{fresh}(r)\), then for any attribute \(a\), \(\sigma, O \models r.a \dashv -\) and \(\psi\) are separated.

**Theorem 2.** For any \(\psi_1, \psi_2, \psi_3\), if \(\psi_1\) and \(\psi_2\) are separated, then \(\psi_1 \ast (\psi_2 \dashv \psi_3) \iff \psi_2 \dashv (\psi_1 \ast \psi_3)\).

**Proof.** The proof is as follows:

1. \((\Rightarrow)\): For any \(\sigma\) and \(O\) such that \(\sigma, O \models \psi_1 \ast (\psi_2 \dashv \psi_3)\), there exist \(O_1, O_2\), such that \(O_1 \ast O_2 = O, \sigma, O_1 \models \psi_1\), and \(\sigma, O_2 \models \psi_2 \dashv \psi_3\). By the definition of \(\ast\), for any \(O_3\) satisfying \(O_2 \perp O_3\),

\[\sigma, O_3 \models \psi_2\) \text{ implies } \sigma, O_2 \ast O_3 \models \psi_3\).

Because \(\psi_1\) and \(\psi_3\) are separated, then by **Lemma 7**, \(\sigma, O_3 \models \psi_2 \dashv (\psi_1 \ast \psi_3)\).

This is \(\sigma, O \models \psi_2 \dashv (\psi_1 \ast \psi_3)\).
2). \((\Leftarrow): \) For any \(\sigma\) and \(O\) that \(\sigma, O \models \psi_2 \rightarrow (\psi_1 \ast \psi_3)\), for any \(O_1\) that \(O_1 \perp O\), if \(\sigma, O_1 \models \psi_2\), then \(\sigma, O_1 \ast O \models \psi_1 \ast \psi_3\). Now we fix this \(O_1\). From \(\sigma, O_1 \ast O \models \psi_1 \ast \psi_3\) we know there exist \(O_2\) and \(O_3\) such that \(O_2 \perp O_1', O_2 \ast O_3' = O_1 \ast O\), \(\sigma, O_2 \models \psi_1\) and \(\sigma, O_3' \models \psi_3\). Because \(\psi_1, \psi_2\) are separated, then \(O_2 \perp O_1\). Thus \(O_3' = O_1 \ast O_3\) for some \(O_3\). Now we have

\[
\sigma, O_2 \models \psi_1, \sigma, O_1 \models \psi_2, \text{ and } \sigma, O_1 \ast O_3 \models \psi_3.
\]

Then we have \(\sigma, O_3 \models \psi_2 \rightarrow \psi_3\), because the choice of \(O_1\) needs no any extra restriction. Thus \(\sigma, O \models \psi_1 \ast (\psi_2 \rightarrow \psi_3)\), because \(O = O_2 \ast O_3\). □

The concept of \textit{separated assertions} is very useful in reasoning OO programs, where we often need to assign several attributes of an object sequentially. \textbf{Lemma 6} and \textbf{Theorem 2} allow us to recombine relative attributes back to an object. Many examples of this kind can be found in Section 6.

Because this logic adopts the classical semantics, it is more expressive than its intuitionistic cousin, e.g., what defined in [25]. We can use it to describe the program state precisely, especially the Opool, i.e., what is or is not in the Opool. For example, the following assertion describe a Singleton Pattern, which requires that there must be at most one object of type \(C\):

\[
\forall r_1, r_2 \cdot \neg \text{fresh}(r_1) \land r_1 : C \land \neg \text{fresh}(r_2) \land r_2 : C \Rightarrow r_1 = r_2.
\]

### 4 Weakest Precondition Semantics of \(\mu\)Java

In this section, based on the storage model given in Section 2.3 and assertion language introduced in Section 3, we define the weakest precondition (WP) semantics for \(\mu\)Java.

#### 4.1 WP Semantics

As usual, the WP semantics of command \(c\) is defined as a predicate transformer, which maps given predicate \(\psi\) to the weakest precondition of \(c\) with respect to \(\psi\). Same as [7], we define semantics only for the well-typed commands, i.e., \(\Gamma, C, m \vdash c : \text{com}\) is supposed true. The static necessities ensured by typing will not appear in semantic rules. Suppose \(\Psi\) is the set of predicates, and \(T = \Psi \rightarrow \Psi\) is the set of predicate transformers, then \([\Gamma; C, m \vdash c : \text{com}] : T\). In most cases, we use \(\lambda\)-notation to give the semantics. We use \(f = g\) in the definition to mean that \(f\) is \(g\). The semantic rules for various structures in \(\mu\)Java are given in Fig. 4, where we use \([\Gamma; C, m \vdash]\) instead of \([\Gamma; C, m \vdash c : \text{com}]\) for conciseness.

The semantics of sequential composition, choice, and iteration are routine, given as rules (SEQ), (COND), and (ITER), where \(\text{fix}(G)\) denotes the least fixpoint of \(G(g) = \lambda \psi \cdot (\neg b \Rightarrow \psi) \land (b \Rightarrow f(g(\psi)))\). The proof of the existence of the least fixpoint can be derived by ordinary procedure.
The semantics of a method invocation, we should consider what is the characteristic predicate transformer with type \( PT \). Following this idea, we define the semantics of a method as a parameterized predicate transformer with type \( PT \cong \text{Ref}^n \rightarrow T \), for some \( n \).
We consider only the non-recursive method here. In fact, we can recast recursive methods to non-recursive ones. We leave the direct treatment of the recursive case as a future work. In rule (MTHD), all local variables are replaced with nil values. This means, on one hand, all local variable are not accessible from outside of the method. It also means that all local variable should be initiated with nil.

For a parameterized predicate transformer \( F : \mathcal{PT} \), we need to apply it to some references \( r_0, r_1, \ldots, r_n \), that stand for the objects referred by \( \text{this} \) and actual arguments, to obtain a predicate transformer \( F(r_0, r_1, \ldots, r_n) \). For convenience, we define an abbreviation that for any expression \( e \),

\[
F(r_0, e, \ldots, r_n) \equiv \lambda \psi \cdot \exists r \cdot e = r \land F(r_0, r, \ldots, r_n)(\psi).
\]

We may also accept more than one expressions in this abbreviation. For example, we can see \( F(r, e) \) in last two rules in Fig. 4.

**Method Invocation**: Based on above definition, we can define the WP semantics for method invocation, given as rule (INV). Here we collect all possible methods of the subclasses and define the weakest precondition as the disjunction of corresponding predicates. Note that \( r : S \) ensures \( r \neq \text{null} \). And we replace \( x \) with \( \text{res} \) in \( \psi \), because the invocation can be viewed as two “actions” sequentially: the first one execute the body of \( v.m(\overline{e}) \) and store the return value in \( \text{res} \), the second save this value into \( x \).

**Object Creation**: Informally, object creation can be thought as two “actions” sequentially: the first one extends the Opool by creating a new object and obtains its reference; the second initiates the object’s state. That is exact the case in OO languages practically, and specified by rule (NEW). Here \( \text{ref}(r, N) \) means that \( r \) refers to an object with type \( N \), which is defined as (suppose \( a_1, \ldots, a_k \) are all attributes of \( N \)):

\[
\text{ref}(r, N) \equiv r : N \land (r.a_1 \Rightarrow) \ast \ast \ast (r.a_k \Rightarrow).
\]

Rule (NEW) asserts that if we append any new object with type \( N \) to current Opool, after the initiation of constructor \( \psi \) will hold.

For convenience, we adopt

\[
\forall \text{fresh}(r) \cdot \text{ref}(r, N) \Rightarrow F(r, \overline{e})(\psi[\overline{r}/x])
\]

as the abbreviation for

\[
\forall r \cdot \text{fresh}(r) \Rightarrow (\text{ref}(r, N) \Rightarrow F(r, \overline{e})(\psi[\overline{r}/x]))
\]

Then the rule (NEW) could be rewritten as:

\[
[\Gamma, N \vdash N : \text{method}] = F
\]

\[
[\Gamma, C, m \vdash x := \text{new } N(\overline{e}) : \text{com}] = \lambda \psi \cdot \forall \text{fresh}(r) \cdot \text{ref}(r, N) \Rightarrow F(r, \overline{e})(\psi[\overline{r}/x])
\]

Readers may ask whether this rule covers the situation of creating an object of class \( N \) with no attributes. The answer is YES. Consider a reference \( r \) refer to an object of type \( N \), and it has no attributes, we have \( \sigma, \emptyset \models \text{ref}(r, N) \), indicating that the new object does not take any space in the Opool. In practice, an empty object is useless. We treat it here as a special and trivial case, and will not discuss it further.
4.2 Properties

Now we prove that the WP semantics for $\mu$Java is well defined, i.e., it forms a well-defined function on all well-typed commands. Additionally, predicate transformers of well-typed commands are monotone functions. We have the following theorems.

**Theorem 3.** Suppose we have built $(\Theta, \Gamma)$ for a program $P$. For any well-typed command $c$ with $\Gamma, C, m \vdash c : \text{com}$, its semantics $[\Gamma, C, m \vdash c : \text{com}]$ is a total function on all formulas. Additionally, if $\Gamma, C \vdash m : \text{method}$, the semantics $[\Gamma, C \vdash m : \text{method}]$ 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.

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

- **Case Skip, Assignment and Return:** Semantics for these commands are direct, so the conclusion holds.

- **Case 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 $F_i = \lambda \text{this} \cdot \lambda \psi \cdot f_i(\psi)[\text{nil}/y]$ is the parameterized predicate transformer for $S_i$. The conclusion holds.

- **Case Object Creation:** Similar to Method Invocation, by the typing rule and hypothesis, we have that there exists $f$ that $F = \lambda \text{this} \cdot \lambda \psi \cdot f(\psi)[\text{nil}/y]$ 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 4.** Suppose we have built $(\Theta, \Gamma)$ for a program $P$. For any well-typed command $\Gamma, C, m \vdash c : \text{com}$, if $\psi$ is a well formed predicate as defined in Section 3, then $[\Gamma, C, m \vdash c : \text{com}]\psi$ is also a well formed predicate.

*Proof.* Straightforward by induction on the structures of $c$. □

**Theorem 5.** Suppose $f : T$ is a predicate transformer produced by WP rules given in Fig. 4, and $\psi, \psi'$ are any well-formed predicates. If $\psi \Rightarrow \psi'$, then $f(\psi) \Rightarrow f(\psi')$.

*Proof.* By induction on the structure of the commands.

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

- **Case Skip:** The proof is trivial.

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

- **Case Assignment II:** By the definition of the separation conjunction and separation implication, the conclusion holds.

- **Case Assignment III:** Since $\psi \Rightarrow \psi'$, then $\psi[r_2/x] \Rightarrow \psi'[r_2/x]$, so we have:

$$
(\exists r_1, r_2 : (v = r_1) \land (r_1.a \leftarrow r_2) \land \psi[r_2/x]) \\
\Rightarrow (\exists r_1, r_2 : (v = r_1) \land (r_1.a \leftarrow r_2) \land \psi'[r_2/x]).
$$
Suppose we have the code in Fig 5, classes $z$ and $e$; the weakest precondition semantics, we have:

Now we give some little examples to illustrate the semantics defined in this section.

### 4.3 Examples

Now we consider the semantics of the iteration in method $f$ hypothesis, we have

Then $F_i(v, \tau)(\psi) \Rightarrow F_i(v, \tau)(\psi')$. By properties of $\land$, the conclusion holds.

### Case Method Invocation

Suppose $\Gamma, S_i \vdash m : \text{method} = F_i = \lambda \text{this}, \bar{\tau} : \lambda \psi : f_i(\psi)[\bar{m}/\bar{\gamma}]$, where $\Gamma, S_i, m \vdash c_i : \text{com} = f_i$. Because $\psi \Rightarrow \psi'$, by induction hypothesis, we have $f_i(\psi) \Rightarrow f_i(\psi')$. Thus $F_i(v, \tau)(\psi) \Rightarrow f_i(\psi')(\bar{m}/\bar{\gamma})$ holds, then $F_i(v, \tau)(\psi) \Rightarrow F_i(v, \tau)(\psi')$. By properties of $\lor$, the conclusion holds.

### Case Object Creation

Suppose $\Gamma, C \vdash N : \text{method} = F = \lambda \text{this}, \bar{\tau} : \lambda \psi : f(\psi)[\bar{m}/\bar{\gamma}]$, where $\Gamma, N, N \vdash c : \text{com} = f$. Because $\psi \Rightarrow \psi'$, by induction hypothesis, we have $f(\psi) \Rightarrow f(\psi')$. Thus $f(\psi)[\bar{m}/\bar{\gamma}] \Rightarrow f(\psi')[\bar{m}/\bar{\gamma}]$ holds, then $F(v, \tau)(\psi) \Rightarrow F(v, \tau)(\psi')$. By properties of $\rightarrow$, the conclusion holds.

#### 4.3 Examples

Now we give some little examples to illustrate the semantics defined in this section.

### Example for Iteration

Suppose we have the code in Fig 5, classes $Nd$ and $Iter$. By the weakest precondition semantics, we have:

$$[\Gamma, \text{Iter}, m \vdash p := p.a : \text{com}] = \lambda \psi : \exists r_1, r_2 : p = r_1 \land r_1.a \leftrightarrow r_2 \land \psi[r_2/p].$$

Now we consider the semantics of the iteration in method $m$, that we must deduce the fix-point of the while loop. To do this, at first, let

$$f(\psi) = [\Gamma, \text{Iter}, m \vdash p := p.a : \text{com}], \quad g(\psi) = \psi.$$

and

$$G(g) = \lambda \psi : (p = \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow f(g(\psi))),$$

$$\text{Fig. 5. Some Simple Examples}$$

Case Method Invocation:

Case Object Creation:
Suppose we have the class declarations as shown in Fig 5, classes A, B, C. Suppose we have the class declarations as shown in [7], due to the absence of object sharing, the semantics of downcast assignment $C\xrightarrow{\text{null}}C$ while $So the semantics of the $\lambda\psi.(p = \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow \exists r_1, r_2 \cdot p = r_1 \land r_1.a \leftarrow r_2 \land \psi[r_2/p])$

$G(G(g)) = \lambda\psi.(p = \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow \exists r_3, r_4, r_2 \cdot r_3 = r_4 \land r_3.a \leftarrow r_4 \land \psi[r_4/p])$

$G(g) = \lambda\psi.(p = \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow \exists r_1, r_2 \cdot p = r_1 \land r_1.a \leftarrow r_2 \land \psi[r_2/p])$

$G(g) = \lambda\psi.(p = \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow \exists r_1, r_2 \cdot p = r_1 \land r_1.a \leftarrow r_2 \land \psi[r_2/p])$

Then we can easily get

$G(g) = \lambda\psi.(p = \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow \exists r_1, r_2 \cdot p = r_1 \land r_1.a \leftarrow r_2 \land \psi[r_2/p])$

So, for describe the fix-point of $G$, we define a predicate $access(r_1, a, r_2)$, which assert that we can reach $r_2$ going forward along field $a$ from reference $r_1$:

$access(r_1, a, r_2) \equiv r_1 = r_2 \lor (\exists r_3 \cdot r_1.a \leftarrow r_3 \land access(r_3, a, r_2))$

With this definition, we can obtain that the fixpoint of $G(g) = \lambda\psi.(p = \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow f(g(\psi)))$ is:

$g = \lambda\psi.(p = \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow \exists r \cdot p = r \land access(r, a, \text{null}) \land \psi[\text{null}/p])$

So the semantics of the while command in method $m$ is:

$[T, \text{iter}, m \vdash \text{while } (p \neq \text{null})\{p := p.a; \} : \text{com}]$

$= \lambda\psi.(p \neq \text{null} \Rightarrow \psi) \land (p \neq \text{null} \Rightarrow \exists r \cdot p = r \land access(r, a, \text{null}) \land \psi[\text{null}/p])$

Example for Reference Types. Suppose we have $C_1 <: C_2 <: C_3$, and $C_1()$ is the constructor of $C_1$. Consider the code in Fig 5, class $\text{Reft}$. One can see that from Theorem 3, we have that method $m$ in class $\text{Reft}$ has a well defined semantics. But in [7], due to the absence of object sharing, the semantics of downcast assignment $a_2 := (C_2)a_3$ is undefined.

Example for Method Invocation. Suppose we have the class declarations as shown in Fig 5, classes $A, B, C$. Suppose $n$ is a method of class $C$. Using the definitions and the reasoning rules of OO Separation Logic, we can reach the following results, where we suppose that class $T1$ has an attribute $f$ of type $T$:

$[T1, n \vdash x := s.m(); y.f := x : \text{com}]$

$= \lambda\psi.\exists r \neq \text{null} \cdot s = r \land r : B \land$

$(\exists r_1, r_2 \cdot s = r_1 \land r_1.b \leftarrow r_2 \land$

$(\exists r_3, r_4 : y = r_3 \land r_2 = r_4 \land (r_3.f \rightarrow * (r_3.f \rightarrow r_4 \leftarrow \psi[r_4/x])))$)

$= \lambda\psi.\exists r \neq \text{null}, r_2, r_3 : s = r \land y = r_3 \land r : B \land$

$r.b \leftarrow r_2 \land (r_3.f \rightarrow * (r_3.f \rightarrow r_2 \leftarrow \psi[r_2/x]))$.

$[T1, n \vdash x := r.m(); z.f := x : \text{com}]$

$= \lambda\psi.\exists r \neq \text{null}, r_2, r_3 : s = r \land z = r_3 \land$

$(\exists r : A \land r.a \leftarrow r_2 \land (r_3.f \rightarrow * (r_3.f \rightarrow r_2 \leftarrow \psi[r_2/x]))) \lor$

$(r : B \land r.b \leftarrow r_2 \land (r_3.f \rightarrow * (r_3.f \rightarrow r_2 \leftarrow \psi[r_2/x])))$. 

4.4 Partial Correctness

Now we define the partial correctness of commands and methods, that shows that we can define a Hoare-style logic framework for µJava based on the WP semantics.

**Definition 2 (Partial Correctness of a Command).** For a command \( c \) with \([\Gamma, C, m \vdash c: \text{com}] = f\), 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 \( \{P\} c \{Q\} \). □

**Definition 3 (Partial Correctness of a Method).** For a method \( m(z) \) of class \( C \) with \([\Gamma, C \vdash m : \text{method}] = F\), and a pair of predicates \( P(this, z) \) and \( Q(this, z) \), if \( P(this, z) \Rightarrow F(this, z)(Q(this, z)) \), then we say that \( m \) is partial correct with respect to precondition \( P \) and postcondition \( Q \). □

We will try to build a full version of Hoare-style proof system for the µJava in the future, based on the WP semantics.

4.5 Frame Rule

Frame rule is extremely important for local reasoning in Separation Logic. It holds in our semantics. Now we prove it here.

**Theorem 6 (Frame Rule).** We have

\[
\{P\} c \{Q\} \quad \text{FV}(R) \cap \text{md}(c) = \emptyset
\]

\[
\quad \{P \ast R\} c \{Q \ast R\}
\]

where \( \text{FV}(R) \) is the set of all program variables (with \( \text{res} \) into account) in \( R \), and \( \text{md}(c) \) denotes the variable set modified by \( c \) with following definition:

\[
\text{md}(c) = \begin{cases} 
\{x\}, & \text{if } c \text{ is } x := \ldots \\
\{\text{res}\}, & \text{if } c \text{ is } \text{return} \ldots \\
\text{md}(c_1) \cup \text{md}(c_2), & \text{if } c \text{ is } c_1; c_2 \\
\text{md}(c_1) \cup \text{md}(c_2), & \text{if } c \text{ is } \text{if } b \text{ c}_1 \text{ else } c_2 \\
\text{md}(c), & \text{if } c \text{ is } \text{while } b \text{ c} \\
\emptyset, & \text{otherwise}
\end{cases}
\]

**Proof.** From \( \{P\} c \{Q\} \), we have \( P \Rightarrow [c]Q \), and we need only to prove \( (P \ast R) \Rightarrow [c](Q \ast R) \). The proof is by induction on the structure of commands. Some notations in Section 4 is used.

**Case Sequential Composition, “c1; c2”:** Suppose \( Q' = f_2(Q) \), we have

\[
\{P\} c_1 \{Q'\}, \quad \{Q'\} c_2 \{Q\}.
\]

Then by induction hypothesis, we have:

\[
\{P \ast R\} c_1 \{Q' \ast R\}, \quad \{Q' \ast R\} c_2 \{Q \ast R\}.
\]

So, by **Theorem 5** we have

\[
P \ast R \Rightarrow f_1(Q' \ast R) \Rightarrow f_1(f_2(Q \ast R)) = f_1 \circ f_2(Q \ast R).
\]

20
Case Condition, “if \( b \) \( c_1 \) else \( c_2 \)”: We have

\[
(P \ast R) \Rightarrow (b \Rightarrow f_1(Q \ast R) \land (\neg b \Rightarrow f_2(Q \ast R)))
\]

\[
\Leftrightarrow ((P \ast R) \Rightarrow b \Rightarrow f_1(Q \ast R)) \land ((P \ast R) \Rightarrow \neg b \Rightarrow f_2(Q \ast R))
\]

\[
\Leftrightarrow (b \Rightarrow (P \ast R) \Rightarrow f_1(Q \ast R)) \land (\neg b \Rightarrow (P \ast R) \Rightarrow f_2(Q \ast R)),
\]

then by induction hypothesis the conclusion holds.

Case Iteration: Treat while \( b \) \( c \) if \( b \) \( c \) else skip, by definition of \( G \) and induction hypothesis we can obtain the conclusion.

Case Skip: The proof is trivial.

Case Assignment I, “\( x := e \)”: From the premise, we have \( P \Rightarrow Q[e/x] \). Because \( R \) does not contain \( x \), we have \( (P \ast R) \Rightarrow (Q[e/x] \ast R) \).

Case Assignment II, “\( v.a := x \)”: From the premise, we have

\[
P \Rightarrow (\exists r_1, r_2 \cdot (v = r_1) \land (x = r_2) \land (r_1.a \mapsto Q'(r_2 \mapsto Q))).
\]

Because \( (v = r_1 \ast R) \Rightarrow v = r_1 \), and \( (x = r_2 \ast R) \Rightarrow x = r_2 \), and by Theorem 2

\[
P \ast R \Rightarrow (\exists r_1, r_2 \cdot (v = r_1) \land (x = r_2) \land (r_1.a \mapsto Q'(r_2 \mapsto Q)) \ast R)
\]

\[
\Rightarrow (\exists r_1, r_2 \cdot (v = r_1) \land (x = r_2) \land (r_1.a \mapsto Q'(r_2 \mapsto Q))).
\]

Case Assignment III, “\( x := v.a \)”: We have

\[
P \Rightarrow (\exists r_1, r_2 \cdot (v = r_1) \land (r_1.a \mapsto r_2) \land Q[r_2/x]).
\]

Considering that \( R \) does not contain variable \( x \), and the properties of operator \( \ast \) and \( \land \), we can easily obtain the conclusion.

Case Assignment By Cast, “\( x := (N)v \)”: We have

\[
P \Rightarrow (\exists r \cdot (r <: N) \land Q[r_2/x]).
\]

Considering that \( R \) does not contain variable \( x \), and the properties of operator \( \ast \) and \( \land \), we know that the conclusion holds.

Case Return, “\( \text{return } e \)”: It behaves as an assignment to res.

Case Method Invocation, “\( x := v.m(r) \)”: We have

\[
P \Rightarrow (\exists r \neq \text{null} \cdot v = r \land (\forall (r : S_i \land F_i(r, \overline{r})(Q[\text{res}/x])))).
\]

Then

\[
P \ast R
\]

\[
\Rightarrow (\exists r \neq \text{null} \cdot v = r \land (\forall (r : S_i \land F_i(r, \overline{r})(Q[\text{res}/x])))) \ast R
\]

\[
\Rightarrow (\exists r \neq \text{null} \cdot v = r \land (\forall (r : S_i \land F_i(r, \overline{r})(Q[\text{res}/x])) \ast R)).
\]

Because \( R \) does not contain \( x \), by the induction hypothesis,

\[
F_i(r, \overline{r})(Q[\text{res}/x]) \ast R \Rightarrow F_i(r, \overline{r})(Q \ast R)[\text{res}/x]).
\]

So, the conclusion holds.
Case Object Creation, ”x := new N(π)”**: By the assumption of frame rule, we have:

\[
P \Rightarrow (\forall \text{fresh}(r) \cdot \text{ref}(r, N) \rightarrow *F(r, τ)(Q[r/x])).
\]

Then

\[
P * R \Rightarrow (\forall \text{fresh}(r) \cdot \text{ref}(r, N) \rightarrow *F(r, τ)(Q[r/x])) * R
\]

By Lemma 9, 10, we have that ref(r, N) and R are separated assertions, then by Theorem 2, we have

\[
P * R \Rightarrow (\forall \text{fresh}(r) \cdot \text{ref}(r, N) \rightarrow *F(r, τ)(Q[r/x])) * R
\]

At last, because R does not contain x, by the induction hypothesis,

\[
Fi(r, τ)(Q[r/x]) * R \Rightarrow Fi(r, τ)((Q * R)[r/x]).
\]

So, the conclusion holds. □

Similarly, we can prove many classical inference rules using the WP semantics. We claim that with the semantics defined in Section 4, the program verification can be reduced to a set of proof obligations in our assertion language.

### 5 Soundness and Completeness Theorems

In this section we prove the soundness and completeness of the WP semantics defined in Section 4. We take \(Ψ\) the space of legal predicates and COM the space of legal commands. We will write \([c]\) instead of \([Γ, C, m ⊢ c : \text{com}]\) when it makes no confusion.

#### 5.1 Soundness

We prove first the soundness theorem. Informally, a WP semantics is sound if the following statement holds for any well-typed command \(c\) and a predicate \(ψ\), if \(c\) executes from a state satisfying the weakest precondition \(ψ'\) of \(c\) with respect to \(ψ\), if \(c\) terminates, the final state will satisfy \(ψ\). It is formally defined as follows.

**Definition 4 (Soundness).** For any given weakest precondition predicate transformer \([•] : \mathcal{T}\), we say it is sound if and only if for any predicates \(ψ, ψ' \in Ψ\) and command \(c \in \text{COM}\) satisfying \([Γ, C, m ⊢ c : \text{com}]ψ = ψ'\), we have: For any pair of states \((σ, O)\) and \((σ', O')\), if \(σ, O \models ψ'\) and \(c, (σ, O) \leadsto^* (σ', O')\), then \(σ', O' \models ψ\). □

**Theorem 7 (Soundness Theorem).** The WP semantics for commands in μJava defined in Section 4 is sound.

**Proof.** We prove the theorem by induction on the structure of commands. We use some notations in WP semantics definition rules, and always assume \(ψ\) the postcondition.
Suppose Case Return, "case return c"; suppose (σ, O) satisfy [c1;c2]ψ, [c1] = f1, and [c2] = f2, thus we have σ, O ⊨ f1(f2(ψ)). Assume that ⟨c1, (σ, O)⟩ ↦* (σ', O'), and ⟨c2, (σ', O')⟩ ↦* (σ'', O''). By induction hypothesis, σ', O' ⊨ f2(ψ), and also σ'', O'' ⊨ ψ.

Case Condition, "if b c1 else c2": Suppose (σ, O) satisfy the precondition, that is, σ, O ⊨ (b ⇒ f1(ψ)) ∧ (¬b ⇒ f2(ψ)). Further, suppose ⟨c1, (σ, O)⟩ ↦* (σ', O'), and ⟨c2, (σ, O)⟩ ↦* (σ'', O''). Then, if [b]⟨σ,O⟩ = rtrue, we have σ, O ⊨ f1(ψ).

By induction hypothesis, we have σ', O' ⊨ ψ, so the conclusion holds. The case when [b]⟨σ,O⟩ = rfalse is similar.

Case Iteration, "while b c": Because while b c is equivalence to if b (c; while b c) else skip, by the definition of G and proof for Condition, the conclusion holds.

Case Skip: The proof is trivial.

Case Assignment I, "x := e": Suppose (σ, O) satisfy the precondition, i.e., σ, O ⊨ ψ[e/x]. By operational semantics rule (O-ASN-I), we have

\( (σ', O') = (σ ⊕ \{ x ↦ σe \}, O) \).

By Lemma 2, we have σ', O' ⊨ ψ.

Case Assignment II, "v.a := x": Suppose (σ, O) satisfy the precondition, i.e., σ, O ⊨ (∃r1, r2 · (v = r1) ∧ (x = r2) ∧ (r1.a ⇔ (r1.a → r2 ∧ ψ(x/r1)))). Then we have there exists r1, r2 such that σv = r1 and σx = r2. From the precondition we have (σv, a) ∈ dom2 O, otherwise r1.a → will be false. So the command does not get stuck. Then by the operational semantics we have

\( (σ', O') = (σ ⊕ \{ (r1.a) → r2 \}) \).

By definitions of * and →*, we can deduce that σ', O' ⊨ ψ.

Case Assignment III, "x := v.a": Suppose (σ, O) satisfy the precondition, that is, σ, O ⊨ (∃r1, r2 · (v = r1) ∧ (r1.a ⇔ r2) ∧ ψ[r2/x]), then we have that there exists r1, r2 satisfying σv = r1, O(r1, a) = r2. From the precondition we have (σv, a) ∈ dom2 O, otherwise r1.a → r2 will be false. So the command does not get stuck. Then by the operational semantics we have

\( (σ', O') = (σ ⊕ \{ x → r2 \}, O) \).

By Lemma 2, we have σ', O' ⊨ ψ.

Case Assignment By Cast, "x := (N)v": Suppose (σ, O) satisfy the precondition, i.e., σ, O ⊨ ∃r · type(r) ∶ N ∧ v = r ∧ ψ[ρ/x]. From ∃r · type(r) ∶ N ∧ v = r, we have type(σv) ∶ N, so by the operational semantic, we have

\( (σ', O') = (σ ⊕ \{ x → σv \}, O) \).

By Lemma 2, the conclusion holds.

Case Return, "return e": Suppose (σ, O) satisfy the precondition, that is σ, O ⊨ ψ[e/res]. By operational semantics we have (σ', O') = (σ ⊕ {res → σe}, O), so σ', O' ⊨ ψ.
Case Method Invocation, "\texttt{\texttt{x} := v.m(\texttt{r})}": Suppose $\sigma, O \models \psi'$, where $\psi'$ is the pre-condition $\exists r \cdot v = r \land \forall r : \exists_i$ $F_i(v, \overline{e}((\psi'[\texttt{res}/x])))$. Then there exists $r$ such that $\sigma v = r$. Since the command is well-typed, there exists an $i$ such that $r : S_i$ and $S_i <: T$. By operational semantics, method $m$ in class $S_i$ is invoked. Suppose $\langle c_i, \{ \texttt{this} \mapsto \sigma v, \overline{e} \mapsto \sigma e, y \mapsto \texttt{nil}, \texttt{res} \mapsto \texttt{nil}, \} \rangle \leadsto^* \langle \sigma', O' \rangle$, where $c_i$ is the body command of method $m$ in $S_i$, we need to show:

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

Assume $\sigma$ does not contain variables $\overline{\xi}$ and $\overline{\eta}$. This can be achieved by renaming local variables involved in $c_i$ (and renaming this as this$'_0$, res as res$'_0$). Let $\sigma_0 = \{ \texttt{this} \mapsto \sigma v, \overline{e} \mapsto \sigma e, y \mapsto \texttt{nil}, \texttt{res} \mapsto \texttt{nil}, \}$, then $\text{dom} \sigma_0 \cap \text{dom} \sigma = \emptyset$, and $\sigma_0$ does not contain variables in $F_i(v, \overline{e}((\psi'[\texttt{res}/x])))$. By $\sigma, O \models F_i(v, \overline{e}((\psi'[\texttt{res}/x])))$ and Lemma 3, we have:

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

Because $F_i = \lambda \texttt{this}, \overline{\xi} \cdot \psi \cdot f_i(\psi) [\texttt{nil}/\overline{\eta}]$, where $F_i(v, \overline{e}((\psi'[\texttt{res}/x])))$ can be viewed as an assertion that replace this, $\overline{\xi}$ and $\overline{\eta}$ in $f_i(\psi)$ to $v$, $\overline{e}$ and $\texttt{nil}$, and we already have $\sigma_0 \texttt{this} = \sigma v$, $\overline{\sigma_0} \overline{\xi} = \sigma e$, $\sigma_0 y = \texttt{nil}$, so by Lemma 2 we have:

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

By inductive hypothesis we have $\langle c_i, (\sigma_0, O) \rangle \leadsto^* (\sigma'_0, O')$. Because $c_i$ does not contain variables in $\sigma$, then by Lemma 1, we have $\langle c_i, (\sigma \cup \sigma_0, O) \rangle \leadsto^* (\sigma \cup \sigma'_0, O')$. Since $[F, S_i, m \vdash c_i : \texttt{com}] = f_i$, by induction hypothesis,

$$\sigma \cup \sigma'_0, O' \models \psi[\texttt{res}/x],$$  

where res denotes the return value. Note that by the operational semantics, the return value is stored in res$'_0$, then

$$\sigma \oplus \{ x \mapsto \sigma'_0 \texttt{res}_0 \} \cup \sigma'_0, O' \models \psi.$$  

Because $\psi$ does not contain variables in $\sigma'_0$, by Lemma 4,

$$\sigma \oplus \{ x \mapsto \sigma'_0 \texttt{res}_0 \}, O' \models \psi.$$  

At last, let us rename variables in $\sigma'_0$ back, then $\sigma \oplus \{ x \mapsto \sigma' \}, O' \models \psi$. The conclusion holds.

Case Object Creation: Suppose $\sigma, O \models \psi'$, where $\psi'$ is the precondition $\forall \texttt{fresh}(r) \cdot \texttt{ref}(r, N) \rightarrow F(r, \overline{e}((\psi'[\texttt{res}/x])))$. Then for any fresh reference $r \notin O$,

$$\sigma, O \oplus \{ \overline{e} \mapsto \overline{e} \} \models F(r, \overline{e}((\psi'[\texttt{res}/x])).$$  

24
Similar to Method Invocation, assume $\sigma$ does not contain variables $z$ and $y$, and let

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

then $\text{dom} \sigma_0 \cap \text{dom} \sigma = \emptyset$, and $\sigma_0$ does not contain variables in $F(v, e)(\psi[r/x])$.

By Lemma 3, we have:

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

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

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

By operational semantics and induction hypothesis, we have

$$\sigma, O' \models \psi.$$
Case Condition, “if $b$ c else $c_2$”: If $[b]_{<\sigma,O>} = \text{rtrue}$, suppose $(c_1, (\sigma, O)) \rightarrow^* (\sigma', O')$ and $\sigma', O' \models \psi$. By induction hypothesis we have $\sigma, O \models f_1(\psi)$, then $\sigma, O \models (b \Rightarrow f_1(\psi)) \land (\lnot b \Rightarrow f_2(\psi))$, that is $\sigma, O \models \psi'$. The case when $[b]_{<\sigma,O>} = \text{rfalse}$ can be proved similarly, so, the conclusion holds.

Case Iteration, “while $b$ $c$” Treat while $b$ $c$ as if $b$ $c$; while $b$ $c$ else skip, by the definition of $G$ and induction hypothesis, the conclusion holds.

Case Skip: The proof is trivial.

Case Assignment I, “$x := e$”: Suppose $(x := e, (\sigma, O)) \rightarrow^* (\sigma \oplus \{x \mapsto \sigma e\}, O)$, and $\sigma \oplus \{x \mapsto \sigma e\}, O \models \psi$. By Lemma 2, $\sigma, O \models \psi[e/x]$. The conclusion holds.

Case Assignment II, “$v.a := x$”: Suppose $(v.a := x, (\sigma, O)) \rightarrow (\sigma, O \oplus \{(\sigma v, a) \mapsto \sigma x\})$, and $\sigma, O' \models \psi$, where $O' = O \oplus \{(\sigma v, a) \mapsto \sigma x\}$. We need to show $\sigma, O \models \psi'$, where

$$
\psi' = \exists r_1, r_2 \cdot (v = r_1) \land (x = r_2) \land (r_1.a \mapsto r_1 \mapsto r_2 \Rightarrow \psi)
$$

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

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

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

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

then by the definition of $\Rightarrow$ and $\Rightarrow$, we have

$$
\sigma, O \models r_1.a \mapsto r_1 \mapsto r_2 \Rightarrow \psi.
$$

So the conclusion holds.

Case Assignment III, “$x := v.a$”: Suppose $(x := v.a, (\sigma, O)) \rightarrow^* (\sigma \oplus \{x \mapsto O(\sigma v)(a)\}, O)$, 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

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

Then, by Lemma 2 we have

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

So the conclusion holds.

Case Assignment By Cast, “$x := (N)v$” Similar to Assignment I, just pay attention that when the command executes successfully and $\text{type}(\sigma v) <: N$. These imply $\sigma, O \models \exists r \cdot \text{type}(r) <: N \land v = r$. So the conclusion holds.

Case Return, “return $e$”: Suppose $(\text{return } e, (\sigma, O)) \rightarrow^* (\sigma \oplus \{\text{res} \mapsto \sigma e\}, O)$, and $\sigma \oplus \{\text{res} \mapsto \sigma e\}, O \models \psi$. Then $\sigma, O \models \psi[\text{res}/e]$. The conclusion holds.

Case Method Invocation, “$x := v.m(\pi)$”: Suppose

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

26
where $\sigma'$ and $O'$ are defined by the operational semantics.

By the type system and operational semantics, the value of $v$ must be a reference $r$ with type $(r) : T$, and $(r)$ contains a method named $m$. So, there exists $i$ that $r : S_i : T$ and $c_i$ is the body command of method $m$ defined in class $S_i$. Assume that $[T, S_i, m \vdash c : \text{com}] = f_i$.

Now, similar to the proof of Theorem 7, let we rename variables in $\sigma'$ to make sure that $\text{dom} \sigma' \cap \text{dom} \sigma = \emptyset$, and let

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

Here $\sigma_0$ does not contain variables in $\sigma$ and $\psi$. And let $\sigma'_0$ be the store satisfying

$$\langle c_i, (\sigma_0, O) \rangle \rightarrow^* (\sigma'_0, O').$$

By hypothesis for Assignment and Lemma 3 we have:

$$\sigma \cup \sigma'_0, O' \models \psi[\text{res}_0/x].$$

By Lemma 1 we also have:

$$\langle c_i, (\sigma \cup \sigma_0, O) \rangle \rightarrow^* (\sigma \cup \sigma'_0, O').$$

Then by hypothesis we have

$$\sigma \cup \sigma_0, O \models f_i(\psi[\text{res}_0/x]).$$

Since $\sigma_0\text{this}_0 = \sigma v, \sigma_0 z = e, \sigma_0 y = \text{nil}$, we can do substitution that:

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

This is equivalent to

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

And because $F_i(\psi)$ does not contains variable in $\sigma_0$, so by Lemma 4, and substitute $x$ with $\text{res}_0$, then:

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

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

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

According to the proof above, we have:

$$\sigma, O \models \exists r \neq \text{null} \cdot v = r \land (r : S_i \land F_i(v, e)(\psi[\text{res}/x]))$$

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

$$\sigma, O \models \exists r \neq \text{null} \cdot v = r \land (\lor (r : S_i \land F_i(v, e)(\psi[\text{res}/x]))).$$

The conclusion holds.
Case Object Creation, \( "x : = \text{new } N(\overline{r})"\): Suppose
\[
\langle x : = \text{new } N(\overline{r}), (\sigma, O) \rangle \rightsquigarrow^* (\sigma \oplus \{ x \mapsto r \}, O'), \text{ and}
\]
\[
\sigma \oplus \{ x \mapsto r \}, O' \models \psi,
\]
where \( O' \) is defined by the operational semantics. Similar to Method Invocation, let
\[
\sigma_0 = \{ \text{this} \mapsto r, \overline{x} \mapsto \sigma e, \overline{y} \mapsto \text{nil} \},
\]
Here we can assume that \( \sigma_0 \) does not contain variables in \( \sigma \) and \( \psi \). Let \( \sigma'_0 \) 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 3 we have:
\[
\sigma \cup \sigma'_0, O' \models \psi[r/x]
\]
Then by induction hypothesis
\[
\sigma \cup \sigma_0, O \oplus \{ \overline{a} \mapsto - \} \models f(\psi[r/x]).
\]
Since \( \sigma_0 \text{this} = r, \overline{x} = \sigma e, \overline{y} = \text{nil} \), we can do substitution that:
\[
\sigma \cup \sigma_0, O \oplus \{ \overline{a} \mapsto - \} \models f(\psi[r/x])[r, \overline{x}, \text{nil}/\text{this}, \overline{y}],
\]
This is in fact \( \sigma \cup \sigma_0, O \oplus \{ \overline{a} \mapsto - \} \models \forall \text{fresh } \sigma \exists \overline{a} F(r, e)(\psi[r/x]) \).

By the operational semantics, we have that \( r \) could be any fresh reference, that is
\[
\sigma, O \models \forall \text{fresh } r \cdot \text{ref}(r, N) \rightarrow F(r, \overline{r})(\psi[r/x]).
\]

This completes the justification of the WP semantics defined in Section 4.

6 Specification and Verification

In this section, we give some illustrative examples to show how the WP semantics can be used to specify and verify OO programs.

6.1 A Queue Example

We study a non-trivial program with some classes that capture many typical OO features. We use the WP semantics to derive semantics of the methods.
class Node : Object {
    Bool value; Node next;
    Node(Bool b){
        this.value = b; this.next = null;
    }
}

class Queue : Object {
    Node head;
    Queue(){
        Node x; x = new Node(false);
        this.head = x;
    }
    Bool isEmpty(){
        Node p; Bool b;
        p = this.head; p = p.next;
        if (p == null) b = true
        else b = false;
        return b;
    }
    Bool enQueue(Bool b){
        Node p, q, n;
        p = this.head; q = p.next;
        while (q != null)
            {p = q; q = q.next; }
        n = new Node(b); p.next = n;
        return b;
    }
}

class EQueue : Queue {
    Node tail;
    EQueue(){
        Node x; x = new Node(false);
        this.head = x; this.tail = x;
    }
    Bool isEmpty(){
        Node p, q; Bool b;
        p = this.head; q = this.tail;
        if (p == q) b = true
        else b = false;
        return b;
    }
    Bool enQueue(Bool b){
        Node p, n;
        p = this.tail; n = new Node(b);
        p.next = n; this.tail = n;
        return b;
    }
}

Fig. 6. Queue and EQueue Example

Suppose we have a Node class as shown in Fig. 6. We want to derive the semantics of it constructor. For convenience, we define a predicate node(n, v, nxt) as follows, which asserts that n holds a Node object, with v and nxt values of its fields.

\[ \text{node}(n, v, \text{nxt}) \equiv \exists r_1, r_2, r_3 \cdot n = r_1 \land v = r_2 \land \text{nxt} = r_3 \land (r_1.\text{value} \rightarrow r_2 \land r_1.\text{nxt} \rightarrow r_3). \]

Then we can have:

\[
[I, \text{Node} + \text{Node::method}]
= \lambda \text{this}, b \cdot \lambda \psi \cdot (\exists r_1, r_2, r_3 \cdot \text{this} = r_1 \land b = r_3 \land
  r_1.\text{value} \rightarrow \ast (r_1.\text{value} \rightarrow r_2 \land
  r_1.\text{nxt} \rightarrow \ast (r_1.\text{nxt} \rightarrow \text{nxt} \land \ast \psi)))) \quad \text{[By Lemma 6 and Theorem 2.]} \]
\[
= \lambda \text{this}, b \cdot \lambda \psi \cdot (\exists r_1, r_2, r_3 \cdot \text{this} = r_1 \land b = r_3 \land
  (r_1.\text{value} \rightarrow \ast (r_1.\text{nxt} \rightarrow \ast) \ast ((r_1.\text{value} \rightarrow r_2 \ast r_1.\text{nxt} \rightarrow \text{nxt} \land \ast \psi))))
\]
\[
= \lambda \text{this}, b \cdot \lambda \psi \cdot \text{node}(\text{this}, - -) \ast (\text{node}(\text{this}, b, \text{null}) \ast \psi). \]

In the second formula above, \( r_1.\text{value} \rightarrow r_3 \) and \( r_1.\text{nxt} \rightarrow \ast \) are separated, thus we can have the deduction. After assignments to several attributes of an object sequentially,
we often need to use Lemma 6 and Theorem 2 to recombine relative attributes back to an object. Many similar examples are in sequel.

**Queue**: Class Queue is implemented via a single linked list with a head node, as shown in Fig. 6. Field `head` holds the list, and the `Node` object referred by `head.next` is the front node of the queue, the last node in the list serves as the rear of the queue.

Similar to `Node`, we define an auxiliary predicate `queue(q, h)` asserting that `q` holds a `Queue` with `h` is the value of `head`.

\[ queue(q, h) \triangleq \exists r_1, r_2 \cdot q = r_1 \land h = r_2 \land r_1.head \leftrightarrow r_2. \]

Then, we derive the semantics of all methods in `Queue` as follows.

**Constructor:**

\[
[\Gamma, \text{Queue} \vdash \text{Queue} \cdot \text{method}] = \lambda \text{this} : \lambda \psi : \exists r_1, r_2, r_3 \cdot \text{this} = r_1 \land r_1.head \leftrightarrow r_2 \land \r_2.next \leftrightarrow r_3 \land (r_3 = \text{null} \Rightarrow \psi[\text{true}/\text{res}]) \land (r_3 \neq \text{null} \Rightarrow \psi[\text{false}/\text{res}]).
\]

**isEmpty:**

\[
[\Gamma, \text{Queue} \vdash \text{isEmpty} \cdot \text{method}] = \lambda \text{this} : \lambda \psi : \exists r_1, r_3 \cdot \text{this} = r_1 \land \r_3.next \leftrightarrow \text{null} \land \psi[r_3, \text{null}/p, q].
\]

**enQueue**: The semantics of `enQueue` is a little complex, because it contains a while loop. We first derive the semantics of the loop.

\[
[\Gamma, \text{Queue}, \text{enQueue} \vdash \text{while} \ (q! = \text{null})\{p = q; q = p.next; \} : \text{com}] = \lambda \psi : (q = \text{null} \Rightarrow \psi) \land (q \neq \text{null} \Rightarrow \exists r_1, r_2 : q = r_1 \land \text{access}(r_1, r_2) \land r_2.next \leftrightarrow \text{null} \land \psi[r_2, \text{null}/p, q]).
\]

Then, let

\[ f(r, b, \psi) = \forall \text{fresh}(r') \cdot \text{ref}(r', \text{Node}) \rightarrow (\text{node}(r', r, \text{null}) \rightarrow (r'.next \leftrightarrow \psi[r'/\text{res}])). \]

we can get the semantics of method `enQueue`.

\[
[\Gamma, \text{Queue} \vdash \text{enQueue} \cdot \text{method}] = \lambda \text{this}, b : \lambda \psi : \exists r_1, r_2, r_3 : (\text{this} = r_1) \land (r_1.head \leftrightarrow r_2) \land (r_2.next \leftrightarrow r_3) \land (r_3 = \text{null} \Rightarrow f(r_2, b, \psi)) \land (r_3 \neq \text{null} \Rightarrow \exists r_4 : \text{access}(r_3, r_4) \land r_4.next \leftrightarrow \text{null} \land f(r_4, b, \psi)).
\]
As shown in Fig. 6, the semantics of `deQueue` in `EQueue` is just the same as in `Queue`. Actually, for a method of `Queue`:

\[ \text{isEmpty}, \text{enQueue} \]

We show how to use the WP semantics to verify methods, taking a `Queue` object with its `isEmpty` and `enQueue` as an example.

We define a predicate `enqueue(a, h, t)` that asserting `q` holds a `EQueue` object with `h` and `t` as field values.

\[ \text{enqueue}(q, h, t) \equiv \exists r_1, r_2, r_3, q = r_1 \land h = r_2 \land t = r_3 \land r_1.\text{head} \iff r_2 \ast r_1.\text{tail} \iff r_3.\]

Then we have:

**Constructor:**

\[ [[\Gamma, E\text{Queue} \vdash E\text{Queue} : \text{method}] = \lambda \text{this} \cdot \lambda \psi : \forall \text{fresh}(r) \cdot \text{ref}(r, \text{Node}) \iff (\exists r_1 : \text{this} = r_1 \land r_1.\text{head} \iff r_2 \land r_1.\text{tail} \iff r_3 \land (r_2 = r_3 \Rightarrow \psi[\text{true}/\text{res}]) \land (r_2 \neq r_3 \Rightarrow \psi[\text{false}/\text{res}]).] \]

**isEmpty:**

\[ [[\Gamma, E\text{Queue} \vdash \text{isEmpty} : \text{method}] = \lambda \text{this} \cdot \lambda \psi : \exists r_1, r_2, r_3 : \text{this} = r_1 \land r_1.\text{head} \iff r_2 \land r_1.\text{tail} \iff r_3 \land (r_2 = r_3 \Rightarrow \psi[\text{true}/\text{res}]) \land (r_2 \neq r_3 \Rightarrow \psi[\text{false}/\text{res}]).] \]

**enQueue:**

\[ [[\Gamma, E\text{Queue} \vdash \text{enQueue} : \text{method}] = \lambda \text{this} \cdot h : \lambda \psi : \exists r_1, r_2 : \text{this} = r_1 \land r_1.\text{tail} \iff r_2 \land (\forall \text{fresh}(r) \cdot \text{ref}(r, \text{Node}) \iff (\text{node}(r, h, t) \iff r_2.\text{next} \iff r_3 \ast r_2.\text{next} \iff r_4 \ast r_1.\text{tail} \iff r_5 \Rightarrow \psi[r_4/\text{res}])).] \]

**deQueue:** The semantics of `deQueue` in `EQueue` is just the same as in `Queue`.

6.2 Verifying Methods

We show how to use the WP semantics to verify methods, taking `Queue` as an example. Actually, for a method of `Queue`, when we want to denote some specification `P` as its precondition, it is necessary that

\[ P \Rightarrow \exists r : \text{this} = r \ast r.\text{head} \iff -. \]
which asserts that before calling the method, this must hold a Queue object. We know that
in the execution of OO program, any method invication has a base object, thus
this part of precondition is true implicitly. Additionally, no method execution can have
effect on this this fact. For simplifying the descriptions, we will omit this part from the
assertions in the following discussions, and use \( P \) as an abbreviation for \( (\exists \cdot \text{this} = \ r \cdot \text{head} \leftarrow \ r) \land P \) in the preconditions.

Considering method is\( \text{isEmpty} \), we specify:

\[
\{ \text{true} \}
\]

\[
\text{is\text{isEmpty}}()
\]

\[
\{ \exists r_1, r_2, r_3 \cdot \text{this} = r_1 \land r_1.\text{head} \leftarrow r_2 \land r_2.\text{next} \leftarrow r_3 \land
(r_3 = \text{null} \Rightarrow \text{res} = \text{true}) \land (r_3 \neq \text{null} \Rightarrow \text{res} = \text{false}) \}
\]

We have derived \( \Gamma, \text{Queue} \vdash \text{is\text{isEmpty}:method}(\text{this})(Q) \) in Section 6.1, thus we
know is\( \text{isEmpty} \) satisfies the specification.

For method de\( \text{Queue} \), let us introduce some predicates:

\[
x = \text{true} \triangleq x = \text{true}\text{true}
\]

\[
x = \text{false} \triangleq x = \text{false}\text{false}
\]

\[
r \leftarrow \text{true} \triangleq r \leftarrow \text{true}\text{true}
\]

\[
r \leftarrow \text{false} \triangleq r \leftarrow \text{false}\text{false}
\]

\[
\text{List} \epsilon r \triangleq r = \text{null}\text{null}
\]

\[
\text{List} (a \cdot a \cdot r) \leftarrow \exists r' \cdot r.\text{value} \leftarrow a \cdot r.\text{next} \leftarrow r' \ast \text{List} \alpha r'
\]

With this predicate, we can specify:

\[
\{ \exists r_1, r_2, r_3, a, \alpha \cdot \text{this} = r_1 \land (r_1.\text{head} \leftarrow r_2 \ast
r_2.\text{value} \leftarrow \ast \ast \text{r}_2.\text{next} \leftarrow r_3 \ast \text{List} (a \cdot a \cdot r_3) \}
\]

\[
de\text{Queue}()
\]

\[
\{ \exists r_1, r_2, r_3, a, \alpha \cdot \text{this} = r_1 \land (r_1.\text{head} \leftarrow r_2 \ast
r_2.\text{value} \leftarrow \ast \ast \text{r}_2.\text{next} \leftarrow r_3 \ast \text{List} \alpha r_3) \land \text{res} = a \}
\]

With the semantics of de\( \text{Queue} \), we can obtain that:

\[
[\Gamma, \text{Queue} \vdash \text{de\text{Queue}:method}(\text{this}, Q)]
\]

\[
= \exists r_1, r_2, r_3, r_4, r_5 \cdot (\text{this} = r_1) \land (r_1.\text{head} \leftarrow r_2) \land (r_2.\text{next} \leftarrow r_3) \land
(r_3.\text{value} \leftarrow r_4) \land (r_3.\text{next} \leftarrow r_5) \land
(r_2.\text{next} \leftarrow \ast \ast \text{r}_2.\text{next} \leftarrow r_5 \ast (Q[r_4/\text{res}]))
\]

\[
= \exists r_1, r_2, r_3, r_4, r_5 \cdot (\text{this} = r_1) \land (r_1.\text{head} \leftarrow r_2) \land
(r_2.\text{next} \leftarrow r_3) \land (r_3.\text{value} \leftarrow r_4) \land (r_3.\text{next} \leftarrow r_5) \land
(r_2.\text{next} \leftarrow \ast \ast \text{r}_2.\text{next} \leftarrow r_5 \ast (Q[r_1, r_2, r_3, a, \alpha \cdot \text{this} = r_1 \land
(r_1.\text{head} \leftarrow r_2 \ast r_3.\text{value} \leftarrow \ast \ast \text{r}_3.\text{next} \leftarrow r_5 \ast \text{List} \alpha r_3) \land \text{res} = a))
\]

\[
= \exists r_1, r_2, r_3, r_4, r_5 \cdot (\text{this} = r_1) \land (r_1.\text{head} \leftarrow r_2) \land
(r_2.\text{next} \leftarrow r_3) \land (r_3.\text{value} \leftarrow r_4) \land (r_3.\text{next} \leftarrow r_5) \land
(r_2.\text{next} \leftarrow \ast \ast \text{r}_2.\text{next} \leftarrow r_5 \ast (Q[r_6, a, \alpha \cdot \text{this} = r_1 \land
(r_1.\text{head} \leftarrow r_2 \ast r_3.\text{value} \leftarrow \ast \ast \text{r}_3.\text{next} \leftarrow r_6 \ast \text{List} \alpha r_6) \land \text{res} = a))
\]

\[
= \exists r_1, r_2, r_3, r_4, r_5 \cdot (\text{this} = r_1) \land (r_1.\text{head} \leftarrow r_2) \land
(r_2.\text{value} \leftarrow \ast \ast \text{r}_2.\text{value} \leftarrow \ast \ast \text{r}_2.\text{next} \leftarrow r_5 \ast \text{List} \alpha r_5)
\]

\[
= \exists r_1, r_2, r_3, a, \alpha \cdot (\text{this} = r_1) \land (r_1.\text{head} \leftarrow r_2) \ast
(r_2.\text{value} \leftarrow \ast \ast \text{r}_2.\text{value} \leftarrow r_3 \ast r_3.\text{value} \leftarrow r_4 \ast r_3.\text{next} \leftarrow r_5 \ast \text{List} \alpha r_5)
\]

\[
= \exists r_1, r_2, r_3, a, \alpha \cdot (\text{this} = r_1) \land (r_1.\text{head} \leftarrow r_2) \ast
(r_2.\text{value} \leftarrow \ast \ast \text{r}_2.\text{value} \leftarrow r_3 \ast \text{List} (a \cdot a) r_3).
\]
Then, method `deQueue` is verified.

### 6.3 Verifying Class Invariants

**Definition 6 (Class Invariants).** We say that predicate `I(this)` is a class invariant of `C` if `I(this)` is established by the constructor of `C`, and preserved by each methods of `C`. Formally, (1) \( \exists \ this = r \land \text{ref}(r, C) \land true \Rightarrow [I, C \vdash C : \text{method}[this, \exists](I(this)) \land I(this) \Rightarrow F(this, \exists)(I(this)) \land I(this) \Rightarrow F(this, \exists)(I(this)) \land I(this) \Rightarrow F(this, \exists)(I(this))] \), assuming \([I, C \vdash m : \text{method}] = F\). □

The first condition says that the only precondition for the constructor is that, before its execution, this has referred to an object, including the raw object; and the constructor will build the invariant on the object.

As an example, we propose following predicate as the invariants for class `Queue`:

\[ I(this) = \exists r_1, r_2 \cdot this = r_1 \land r_1, \text{head} \leftarrow r_2 \land \text{access}(r_2, \text{next}, r_\text{null}). \]

To verify that `I` is a class invariant of `Queue`, we have the following deductions:

\[
[I, \text{Queue} \vdash \text{Queue : method}[\text{this}, I(\text{this})] = \forall \text{fresh}(r) \cdot \text{ref}(r, \text{Node}) \rightarrow (\text{node}(r, -, -) \land \text{access}(\text{queue}(\text{this}, -) \land \text{r}_1, \text{head} \rightarrow \text{r}_2 \land \text{access}(\text{r}_2, \text{next}, \text{r}_\text{null}))))]
\]

This is just \( \exists r : this = r \land \text{ref}(r, \text{Node}) \land true \), so we have `I(this)` is established by the constructor.
We use true as the precondition of \textit{enQueue}, by

\[
[I, \text{Queue} \vdash \text{enQueue} : \textbf{method}](\textbf{this}, I)
= \exists r_1, r_2, r_3 : (\textbf{this} = r_1) \land (r_1.\text{head} \leftrightarrow r_2) \land (r_2.\text{next} \leftrightarrow r_3) \land \\
(r_3 = \text{null} \Rightarrow f(r_2, b, I)) \land (r_3 \neq \text{null} \Rightarrow \exists r_4 : \text{access}(r_3.\text{next}, r_4) \land \\
r_4.\text{next} \leftrightarrow \text{null} \land f(r_4, b, I))
\]

we have that \textit{enQueue} preserves $I$.

At last,

\[
[I, \text{Queue} \vdash \text{deQueue} : \textbf{method}](\textbf{this}, I)
= \exists r_1, r_2, r_3, r_4, r_5 : (\textbf{this} = r_1) \land (r_1.\text{head} \leftrightarrow r_2) \land (r_2.\text{next} \leftrightarrow r_3) \land \\
(r_3.\text{value} \leftrightarrow r_4) \land (r_3.\text{next} \leftrightarrow r_5) \land \\
(r_2.\text{next} \leftrightarrow * (r_2.\text{next} \leftrightarrow r_3 \land \text{List}(a \cdot \alpha) r_3),
\]

we have:

\[
P = \exists r_1, r_2, r_3, a, \alpha : \textbf{this} = r_1 \land r_1.\text{head} \leftrightarrow r_2 \land r_2.\text{next} \leftrightarrow r_3 \land \text{List}(a \cdot \alpha) r_3,
\]

we have:

\[
P \Rightarrow \exists r_1, r_2, r_3 : \textbf{this} = r_1 \land r_1.\text{head} \leftrightarrow r_2 \land r_2.\text{next} \leftrightarrow r_3 \land \text{access}(r_3.\text{next}, \text{null})
\]

\[
\Rightarrow [I, \text{Queue} \vdash \text{deQueue} : \textbf{method}](\textbf{this}, I).
\]

So, method \textit{deQueue} preserves $I$, this concludes our verification that $I$ is an class invariant of \textit{Queue}, as we know that the only other method \textit{isEmpty} does not modify the object.

Of course, the example is still simple. If we want to verify complicated programs with specifications about pre and post conditions and invariants, we must develop advanced techniques, such as Abstract Predicate [23], to write proper specification. We will embark on this topic in future.

\section{Conclusions and Future Work}

Based on an extended storage model, we presented a revised Separation Logic, OO Separation Logic, as a assertion language for OO programs. Recognizing the key role played by weakest precondition (WP) semantics in classical theories [12, 19], with the
support of a typing environment, we developed a WP semantics for an model OO language \(\mu\)Java, and proved that the WP semantics is both sound and complete. In addition, some properties of OO Separation Logic and WP semantics are proved, including properties relative to an useful new concept of separated assertions specially to OO Separation Logic. Hoare triples and WP based verification for OO programs are introduced, and the frame rule is proved. Conducting a comparison to existing work [7, 11, 25], we could conclude that our semantics captures the essentials of object-orientation in a more adequate and useful way. Further, we showed, by some examples, that the semantics could be used in specifying and verifying OO programs, and articulated with major popular verification techniques.

As for the future work, we are keen to address both theoretical and practical aspects. It would be interesting to study the properties and inference rules of OO Separation Logic, paving the way for more effective reasoning. We take also interest in the connection between our logic and the Separation Logic. On the practical direction, it is important to extend the languages to support specifications in order to develop modular verification techniques [20, 24]. The nature of WP semantics enables us to define the refinement relations between programs, which could benefit the verification of program transformations [13]. Further, given the availability of both our operational and weakest precondition semantics, it is feasible to define data refinement so that we can study the refinement relationship between programs/specifications at different abstract levels, therefore provide the possibility of programming from specifications [19] or even code generation.

References


36