A Weakest Precondition Semantics for OO Languages:
An OO-Separation Logic Approach

Quan Long, Qiu Zongyan, and Wang Shuling
LMAM and Department of Informatics,
School of Mathematical Sciences, Peking University
{longquan,qzy,joycy}@math.pku.edu.cn

Abstract. In recent years, many researchers in the programming language and formal methods communities have been investigating weakest precondition (WP) semantics for object-oriented (OO) programs. Based on a modified version of Separation Logic, OO Separation Logic, we develop in this article a WP semantics for an OO language with most important object-oriented features including subtypes, visibility, inheritance, dynamic binding and reference types. Giving a clear comparison to existing work, we conclude that the WP semantics defined here captures the essentials of object-orientation. Further, in the WP semantic model, we define program transformation in terms of refinement. With some case studies, we show that, supported by the semantics defined, it is easier to model many practical program transformations in a reasonable way.

keywords: Object Orientation, Weakest Precondition, Separation Logic, Semantics, Refinement

1 Introduction

In the communities of both software development and programming languages, it is evident that object-orientation (OO) is and will remain an important concept. The languages which support OO concepts give a level of abstraction, separating the view of what a software component does from the details of how it does. It is also clear that certain features of objects, say, inheritance, object references, and dynamic binding bring great benefits to the development, reuse and maintenance of software products.

There have been extensive studies of formal systems in conjunction with OO languages. For instances, America and de Boer presented a logic for a parallel language POOL [2], an imperative language with object sharing, but no subtyping and method overriding. Abadi and Leino defined an axiomatic semantics for an imperative OO language with object sharing [1], which lacks dynamic binding of method invocation. J. He, Z. Liu, X. Li and S. Qin proposed a calculus for an OO language [16], which covers most of important features of sequential Java using Hoare and He’s Unifying Theories of Programming [17].

The list of existing important literatures on formal modeling OO programs for reasoning and verification (e.g. [23, 33]) is too long to be exhausted. What we concern most in this article is the extension of weakest precondition (WP, originated to E.W. Dijkstra [11]) semantics for OO languages. In a WP semantic model, a piece of code is considered as a predicate transformer which transfers a given predicate to the weakest
precondition which the code requires. It is well-known that WP-semantics is a powerful technique in program reasoning, verification, validation, and also, refinement. Besides many successful cases, as a recent example, Leino showed in [20], that the ESC/Java technique [12] is in fact the technique of weakest preconditions with additional use of a certain WP property that holds only for a restricted class of programs.

Based on the WP semantics, C. Morgan showed a refinement calculus for reasoning across specifications and programs in his book [22]. However, the calculus only works for ordinary imperative languages, such as Pascal or C without pointers.

According to our knowledge, researchers have attempted to extend WP semantics for OO since 1999. In [10], F.S. de Boer proposed a WP calculus for OO programs and claimed to support a subset of object sharing (or aliasing). Besides many restrictions to the programs and assertions, the main problem in the article is that the formalism for expressing the semantics is restricted to the form of syntactic substitution (in conjunction with some accessor definitions). For example, the substitution for assignments is defined as follows:

\[
q.x[e/x] = \begin{cases} 
\text{def} & \text{if } (q[e/x]) = \text{self} \\
\text{else} & (q[e/x]).x 
\end{cases}
\]

where “\( q = \text{self} \)” stands for that \( q \) refers to current object.

With this technique, paid many prices of discussing special cases the author imagined\(^1\), the WP semantics is defined syntactically. But actually, for a program, it is impossible to check “\( q = \text{self} \)” syntactically. This makes the efforts of defining WP semantics syntactically meaningless. The similar problem appeared in C. Pierik and F.S. de Boer’s later work [28] which attempts to give a pure syntax-based Hoare Logic for OO programs.

A. Cavalcanti and D. Naumann made a significant contribution to give the WP semantics for OO languages [6, 7]. They defined an OO language with subtyping, polymorphism, dynamic binding, but no sharing. Supported by a typing environment, each command in the language has a semantics as a predicate transformer. The notion of OO refinement is defined too.


These articles are significant progresses of WP semantics for object-orientation. But the main problem of the work is that the basic semantic models adopted (e.g., in [6, 7]) are not based on object references which is an essential feature of popular OO languages such as Java, C++, and C#. So object sharing and updating can not be treated. An explicit example is that in the OO language defined in [7], only WP semantics of assignment with upcast can be defined. It is due to, as far as our opinion, their neglecting of object sharing. In [5] and [9], we can not find any refinement law related to references or pointers. Further, in [9], the authors even made some mistakes about refactoring

\(^1\) Unfortunately, we found an example which is permitted by his restrictions but failed to be included in the definition. Although it is easy to be fixed, the fact shows that it is not easy to manage the semantics in such a way.
laws when they tried to encode some reference-related refactorings in the non-reference semantic model. Thus, in one sentence, using a non-reference semantic model, it is difficult to verify many of OO concerns which are interested by industry. We would like to remedy these problems in this work.

Separation Logic, developed by J.C. Reynolds, P.W. O’Hearn et al [18, 32], is a powerful calculus to verify the properties of programs with shared mutable data structures involving pointers. Most of work related to the logic focused on lower level, C-like languages. For instance, a WP semantics of a relatively low-level language was given in [18]. Recently, P.W. O’Hearn, H. Yang and J.C. Reynolds added static modularity to Separation Logic, where the internal resources of a module are hidden from the clients using the hypothetical frame rule [25]. However, as pointed out by M. Parkinson and G. Bierman in [27], “their work is severely limited as it only models static modularity... Hence, it can not be used for many common forms of abstraction, including ADTs and classes, where we require multiple instances of the hidden resource.” Parkinson and Bierman addressed the problem of writing specifications for programs with various forms of modularity, including procedures and Java-like classes, and introduced a new notion of abstract predicates which is flexible in reasoning about different forms of abstraction found in modern languages. However, the model did not offer full support of many OO features. Furthermore, there is not a type system to separate the concerns of static and dynamic verification. Another issue we concern is that their model is not based on weakest preconditions, and thus they did not gain the completeness property. Nevertheless, the work mentioned above enlighten us to use a lower level assertion language to define the WP semantics for OO languages. Unfortunately, we realize that it might not be easy to model shared mutable data in OO programs using the original Separation Logic. So we revised it and reached an OO Separation Logic finally, which will be shown in the following.

In this article, we developed a weakest precondition semantics which supports reference type, or object sharing, based on an OO Separation Logic. We use an Object Pool, Opool for short to replace the heap in the original Separation Logic. The basic unit in Opool is the reference-object pair, while the references can be shared by not only names in program texts but also attributes of object instances. Based on these, we defined the semantics of an OO language with subtypes, visibility, polymorphism, inheritance, dynamic binding and reference types. The language is sufficiently close to the sequential part of Java and can be used in meaningful case studies to capture many central challenges in modeling OO programs. Using this semantics, we can advance the theory of [5] by establishing more reference-related refinement laws for OO programs. In our model, every instance is an object and every variable (and every attribute in an object) is a reference. So it is a pure object model. And also, we have a pure reference semantics rather than the value semantics as in [18, 32, 27], etc. This makes it very convenient when one reasons across OO programs.

Another contribution is the definition of refinements which acts as an essential concept in the formal method for correct development of software. The authors of [7] and [16] have given the definitions of refinement based on their semantic models. Why another one? Besides the theoretical reason — our definition is based on a quite different semantic model, the practical reason is also obvious. Both definitions in [7] and [16]
bind class declarations and main programs together. This is too restricted. Practically, class declaration part of a program is often developed separately from the main program, and they can even have different life time. Theoretically, when one tries to formalize the refactoring rules in [13] based on the refinement definitions proposed there, she or he has to manage many conditions the program should satisfy to make the change of class declarations consistent to the main program. In [9] and [21], one can see many side conditions of these kinds are listed. The side conditions are so many that it might make readers worry whether there is any other condition missed. In this article, we defined the refinement based only on the class declarations with no requirement on main program. We will illustrate the advantage via some examples.

Other main contributions of this work are as follows: We handle the dynamic binding issue by fixing the code of a certain method to its class. According to our knowledge, the existing work on the formal semantics of non-trivial OO languages either did not handle it [1, 26, 31, 27], or handled it by looking up the method body during execution [7, ?]. Furthermore, different from many existing work, for instances [1, 7, 16], we have constructor here which makes the language closer to the real programming languages. Thus we could formalize many OO design pattern-directed refactoring rules (e.g. Abstract Factory in [14]) more clearly, rather than encode constructors using other methods.

The rest of this paper is organized as follows. We give a brief overview of the original Separation Logic in Section 2, detailing the features concerned in this article. Our revised version, an OO Separation Logic is presented in Section 3. We introduce an OO language in Section 4. In Section 5, the static environment system including the type judgement and method lookup is proposed. The weakest precondition semantics is presented in Section 6. In Section 7, we give the definitions of refinement and some examples as well. In the last section, the article is concluded and some future research directions are discussed.

2 An Overview of Separation Logic

Separation Logic [32, 18], derived from Bunched Implication (BI) logic [24, 29], is an extension of Hoare Logic. Aiming to capture the low level characters of machine language, it is specially designed for reasoning about shared mutable data structures. In Separation Logic, heaps are added to the storage model to represent the mutable structures. It supports reasoning about different portions of heaps which can be combined in a modular way using the separating conjunction operator.

2.1 The Storage Model

In the storage model of Separation Logic, a store is a partial mapping from variables to values, and a heap is a partial mapping from addresses to values. Based on sets Values and Addresses chosen, and the set Variables, we have:

\[
\text{Store} \triangleq Variables \rightarrow Values \\
\text{Heap} \triangleq Addresses \rightarrow Values
\]
A state is a pair consisting of a store and a heap \((s, h)\), i.e.,

\[
\text{State} \overset{\text{def}}{=} \text{Store} \times \text{Heap}
\]

To talk about separation properties, \(h_1 \perp h_2\) indicates that \(h_1\) and \(h_2\) have disjoint domains, i.e., \(\text{dom } h_1 \cap \text{dom } h_2 = \emptyset\), and \(h_1 \cdot h_2\) indicates the union of the two heaps, provided that \(h_1 \perp h_2\).

### 2.2 Assertions

To verify the properties of programs, besides the ordinary predicates used in Hoare logic, some additional assertion forms on the states are introduced in Separation Logic.

Basic assertion \(\text{emp}\) is used to denote that the heap is empty. Precisely, \(\text{emp}(s, h)\) iff \(\text{dom } h = \emptyset\). The singleton heap containing address \(e\) and content \(e'\) is expressed by \(e \perp e'\) with the precise definition

\[
\{ e \mapsto e' \}(s, h) \text{ iff } \text{dom } h = s(e) \wedge h(s(e)) = s(e')
\]

where \(s(e)\) stands for the evaluating result of \(e\) in store \(s\) which is denoted by \([e]\)\text{exp}\s in [32].

The central notion in Separation Logic is the \textit{Separating Conjunction} operator \(\&\). For two predicates \(p\) and \(q\), \(p \& q\) asserts that the heap can be split into two disjoint parts in which, \(p\) holds for one while \(q\) holds for the other.

\[
\{ p \& q \}(s, h) \text{ iff } \exists h_1, h_2 \bullet h_1 \perp h_2 \wedge h_1 \cdot h_2 = h \\
\wedge [p](s, h_1) \wedge [q](s, h_2)
\]

In general, heaps of more than one element can be specified by using \(\&\) to combine disjoint smaller heaps.

Another important operator is the \textit{Separating Implication} operator \(\Rightarrow\). It is very useful in backward reasoning. Assertion \(p \Rightarrow q\) says that, if the current heap is extended with any disjoint part in which \(p\) holds, then \(q\) will hold in the extended heap. The definition is as follows:

\[
\{ p \Rightarrow q \}(s, h) \text{ iff } \forall h' \bullet h' \perp h \\
\wedge [p](s, h) \implies [q](s, h')
\]

The key to these operators is that \(\&\) decomposes the current heap into pieces and \(\Rightarrow\) talks about the extension of the heap by new locations. It has been proven that these operations provide an effective way for local reasoning. Without them, when an alteration occurs in a single heap that affects the meaning of pointer \(p\), we must know that it will not affect another pointer \(q\), and even further other pointers. This makes it indispensable to check the aliases. Furthermore, with the frame rule introduced in [32], the \textit{Separating Conjunction} makes us able to concentrate only on the just heap cells which are altered in the program execution, while others remain unchanged. These features can help us to reason about programs as local as possible [30].

There are many useful abbreviations introduced. For instance, \(x \mapsto \_\) is written to denote that \(\exists e \bullet x \mapsto e\). Another example is \(e \mapsto e'\) which stands for \(e \mapsto e' \& \text{true}\). This is very useful when one would like to express that some additional property holds for a certain location of the heap.
3 Syntax of the Language

The core object-oriented language investigated in this article is similar to a subset of sequential Java. What we are mainly concerned with is the essential OO features relating to object sharing, updating, and creation. So we do not have the command while which can be handled with fix point of a certain equation.

The syntax of expressions, predicates and commands is as follows:

\[
\begin{align*}
    e & ::= \text{true} | \text{false} | \text{self} | \text{null} | le \\
    le & ::= x | x.a | \text{self}.a \\
    b & ::= e = e | e \land b | e \lor b \\
    \psi & ::= \psi \Rightarrow \psi | \psi \lor \psi | \psi \land \psi | \forall t : T \bullet \psi | \exists t : T \bullet \psi \\
    C_1 & <: C_2 | \text{emp} | e \mapsto a | \psi * \psi | \psi \mapsto \psi \\
    c & ::= \alpha : [\psi, \psi] | \text{if} b \text{ else } c \\
    | c; c | le ::= e | le ::= (C)e | le.m(\tau) | \text{self}.m(\tau) | C.\text{new}(x, \tau) | \text{free}(le)
\end{align*}
\]

Here are some explanations about these structures:

- The language is similar to a sequential subset of Java (except for the free command). We do not permit the declaration of local variables in method body and methods have no returned values. Because those features can be encoded by the parameters of methods, thus, are not essential restrictions.
- We use le to denote the expressions which can appear on the left hand side of assignments, and use \(a, x\) to denote attributes of classes, parameters of methods respectively. According to the restricted syntax, the expression \(e.a.a\) is illegal. However, those features can be encoded by assignments. For instance, one can let \(x = e.a\) and then refer to \(x.a\). Thus, the simplification is not an essential restriction.
- We consider the cast as part of a command rather than an expression. This is also a non-essential restriction. And further, we will give the semantics of cast related assignments in WP framework, which is impossible in the framework of [7].
- The assertions \(\psi\) includes the form \(\psi \Rightarrow \psi, \psi \lor \psi, \psi \land \psi\) with the meaning introduced in Section 3. This is the key point why we can solve the problem that [7] can not — we use a lower level assertion language.
- Similar to [22] and [7], we have specifications in the command part of our syntax to support the development from abstract specifications to executable code.
- The execution of \(C.\text{new}(x, \tau)\) will create an object initiated with parameter \(\tau\) and let \(le\) refer to it.
- Similar to the language in [1], to keep the semantics simple and clear, we have boolean type as the only primitive type. Hence, true and false are the only primitive values. We also consider boolean as a reference type. This will be explained in detail later.

Similar to Java’s assumption, if it exists, we consider main() as a special method where the program starts. So we do not have the special type program [7] in the typing environment for the main programs. A program is a sequence of class declarations in
which there is zero or one class containing a method \texttt{main()} which stands for the main program as the starting point of the execution. The syntax of the program is as follows.

\[
\begin{align*}
\text{Program} & ::= \text{cds} \\
\text{cds} & ::= \emptyset \mid \text{cd} ; \text{cds} \\
\text{cd} & ::= \text{class } C \text{ extends } C \{ \\
& \quad \text{pri } a : T; \\
& \quad \text{pro } a : T; \\
& \quad \text{pub } a : T; \\
& \quad \text{C}(	ext{para})\{e\}; \\
& \quad m(\text{para})\{e\} \\
& \}
\end{align*}
\]

As shown, the program is considered as a group of classes in which each class offers a group of methods as the services to client code. There are three restrictions for the programs:

- We assume a pre-defined class \texttt{Object} as the super class of all user defined classes, and \texttt{bool} is the only primitive type (pre-defined) which is not super- or sub-type of any other classes.
- If there is a \texttt{main()} method as the starting point of the system, it should be contained in the last class declaration.
- As the notation implies, the constructor \texttt{C(para)}\{e\} must have the same name with the class and further, we suppose that \( e \) in a constructor is a sequence of assignments with the form \texttt{self.a := e}, which initialize the state of the newly created object.
- We have method overriding but do \textit{not} permit method overloading in class declarations.

4 \hspace{1em} An OO Separation Logic

For supporting the important concepts related, most OO languages adopts the reference model, in which the value of every variable is a reference to an object in an object pool, and the value of every attribute of an object is also a reference to an object. (A special case is that the value can be null to mean that the variable or attribute refers to no object.) In this storage model, there is more possibility of sharing: In addition to that different names (variables) can share the same reference as their value, in the object pool, different attributes of objects can share the same reference. And also, attributes can have sharing with variables.

It seems difficult to model these sharing by the original Separation Logic. Thus we revise the logic for OO applications. We begin with the revised storage model, and then the assertions.

4.1 \hspace{1em} The Storage Model

In the storage model, we have two basic sets:
– **Name**: The set of names. They can be used as the variable name or the attribute name in the program.

– **Ref**: The infinite set of references which can be thought as the addresses of objects. There is a special and distinguishable reference $r_{null}$ which never refers an object.

A state here consists of two sets (mappings). They are:

1. $\sigma : \text{Name} \rightarrow \text{Ref}$
   The store $\sigma$ maps names (variables) to their references. The state is modified by assignments. For instance, after the assignment $x := y$, $\sigma_x$ will be changed to the same value as $\sigma_y$.

2. $O : \text{Ref} \times \text{Name} \times \text{Ref}$
   For a triple $(r, n, r_1) \in O$, $r$ is the reference to a certain object $o$, $n$ is the name of an attribute of $o$, and $r_1$ is the reference which is the value of the attribute $n$ of $o$. In fact, $O$ is a multi-value function. If $r$ is a reference, we define:
   
   $$O(r) = \{(n, r_1) \mid (r, n, r_1) \in O\}$$

   $O(r)$ is a set of pairs, representing the object referred by $r$. We will sometimes use letter $o$, possibly with subscripts, to indicate objects. We will call $O$ an object pool, or Opool for short.

   In our model, for any $r \in \text{dom}(O)$, $O(r)$ is a further finite function which maps the attribute names of the object referred by $r$ to the references those are the values of these attributes. We define two projects: $O(r)_1$ and $O(r)_2$. There meaning are as follows: $O(r)_1$ is the set of attributes of the object referred by $r$, and $O(r)_2$ is the set of corresponding values (references).

   We assume that in the representation of the objects in $O$, every object has a special attribute named $\text{type}$, indicating the type which the object belongs to. In this case, we can easily determine the type of the objects at any time. Because this feature does not have critical impact on the discussion of this paper, we will omit the treatment of it, and simply assume that we can get the type of an object directly.

   Please notice the difference between the storage model of OO Separation Logic and the traditional Separation Logic. In the Separation Logic, all the shared mutable data structures are stored in the heap, while in our logic the shared mutable data are in Opool. They play the similar role. However, our logic adapts a pure reference semantics rather than pointer semantics. Hence our storage model is more abstract than the traditional one.

   Most of the operations in the program will keep the domain of $O$ unchanged, except the object creations and destructions. For instance, if the user creates an object $o$ and lets variable $x$ refer to it, then the system will take a fresh reference $r$, and let $O' = O \oplus \{r \mapsto o\}$ and $\sigma'x = r$. Here we use $O'$ and $\sigma'$ to denote the Opool and the store after the operation. If an object $o$ is destructed via variable $x$ when $\sigma x = r$, then after the operation, we will have $O' = O\setminus\{r \mapsto O(r)\}$, and $\sigma'x = r_{null}$, i.e., $x$ refers to an empty object. The operators $\oplus$ and $\setminus$ denote the standard override and set difference operations, respectively.

   Assignments to attributes change the range of $O$. If $x$ refers to object $o$, i.e., $O(\sigma x) = o$. After the assignment $x.a := y$, we will have $o(a) = \sigma y$, that is, $O(\sigma x)(a) = \sigma y$. 

8
We assume that there is a special name null ∈ Name, with that σnull = rnull holds forever. As we stated before, rnull refers to no object. But for convenience of assertion writing, we assume O(rnull) = ∅ which denotes an empty object.

To state the separation properties, O₁ ⊥ O₂ indicates that Opools O₁ and O₂ have disjoint domains, i.e., dom O₁ ∩ dom O₂ = ∅, and O₁ · O₂ indicates the union of them when O₁ ⊥ O₂. In the computational model of our logic, a state is a pair of a store and an Opool, i.e.,

State = \{⟨σ, O⟩ | σ and O defined above\}

4.2 Assertions

To be consistent to the Separation Logic, we keep the assertion language syntactically unchanged except that a written form for objects is adopted. The semantics of assertions is revised here.

To unify notations, we define first an auxiliary function γ<σ,O>e to extract the references from expressions:

\[
γ_{<σ,O>}e \overset{\text{def}}{=} \begin{cases} 
σe & \text{if } e \in \text{Name} \\
O(γ_{<σ,O>}e')(a) & \text{if } e = e', a \land e' \in \text{Name} \land a \in O(γ_{<σ,O>}e') 
\end{cases}
\]

The semantics of assertions is defined on states as follows.

**BASICS:**

The assertion emp is true only on the empty Opool:

\[
\text{emp}(σ, O) \text{ iff } O = ∅
\]

For the semantics of the singleton assertion, a notation is necessary to represent objects. We use \{n₁ : r₁, . . . , n_k : r_k\} to represent an object with attribute names n₁, . . . , n_k while the corresponding values of these attributes are r₁, . . . , r_k respectively. We will use o, possibly with subscription, to denote the representation of an object. The singleton assertion here takes the form of e ↦ o where o is an object. The semantics of it is defined in two cases as follows:

\[
\{e \mapsto o\}(σ, O) \text{ iff } σ(x) = \text{dom } O \land O(σ(x)) = o
\]

\[
\{e.a \mapsto o\}(σ, O) \text{ iff } \exists O', o' \cdot O'(γ_{<σ,O'}>e) = o' \land o'(a) = \text{dom } O \land O(o'(a)) = o
\]

With the support of the function γ, this law covers all the cases involving variables and attributes.

For the central operators ∗ and →, our definitions separate the Opool, with a similar form to the Separation Logic. The semantic definitions of them take the same forms as in Separation Logic:

**SEPARATING CONJUNCTION:**
\[
[p \cdot q](\sigma, O) \iff \exists O_1, O_2 \cdot O_1 \perp O_2 \wedge O_1 \cdot O_2 = O \\
\wedge [p](\sigma, O_1) \wedge [q](\sigma, O_2)
\]

\(p \cdot q\) holds iff \(p\) and \(q\) hold on two separated parts of the original Opool, respectively.

**SEPARATING IMPLICATION:**

\[
[p \rightarrow q](\sigma, O) \iff \forall O_1 \cdot O_1 \perp O \wedge [p](\sigma, O_1) \\
\implies [q](\sigma, O_1 \cdot O)
\]

\(p \rightarrow q\) holds iff the current Opool is extended with a disjoint part in which \(p\) holds, then \(q\) will hold in the extended Opool.

In our logic, many existing properties, e.g., the frame rule, of traditional Separation Logic also hold.

## 5 Static Environments

In [7], the authors developed a static environment to localize and simplify the definition of the WP semantics. What we do here is similar to theirs. However, our static environment is consisted of two components: \(\Gamma\) and \(\Theta\), where \(\Gamma\) stands for typing environment, i.e. the structure of the classes, and \(\Theta\) for the method lookup environment. They both are established by scanning the programs before execution.

### 5.1 Typing Environment

The typing environment \(\Gamma_{cds}\) is a system which records the static structural information of the class declarations \(cds\) with which the types of all expressions and the well-formedness of predicates and commands can be deduced. We often abbreviate it as \(\Gamma\) when there is no confusion. It is a tuple as follows:

\[
\langle \text{cname}, \text{super}, \text{method}, \text{attr}, \text{locvar} \rangle
\]

Except for \text{cname}, the other elements are relations over the classes, methods, and attributes:

- **cname**: The set of all the class names which appear in the class declarations \(cds\).
- **super**: A relation which maps each class to its immediate superclass. \text{super}(C_1, C_2) stands for that \(C_2\) is the immediate superclass of class \(C_1\). We will use \(C < C_1\) as the abbreviation of \((C, C_1) \in \Gamma.\text{super}\), i.e., \(C_1\) equals to \(C\), or is a super class of \(C\).
- **method**: A relation which relates each class to its method signatures, where \text{method}(C, m([\text{param}])) stands for the fact that \(m([\text{param}])\) is a method signature of class \(C\). Please notice that the constructors are also included in \text{method}.
– **attr**: A relation over classes, methods and attributes. $\text{attr}(C, m, a : T)$ means that in class $C$, attribute $a$ with the declared type $T$ is visible for method $m$. As a shorthand, we use $\text{attr}(C, a : T)$ to mean that in class $C$, attribute $a$ is visible for every method $m$ which satisfies the relation $\text{method}(C, m, (\text{param}))$ for some param.

– **locvar**: A relation over classes, methods and attributes. It is combined by all the parameters in methods signatures. $\text{locvar}(C, m, x : T)$ means $x$ (with declared type $T$) is a parameter of method $m$ in class $C$.

### Construction of $\Gamma$.

In this subsection, we will present the rules for the construction of $\Gamma$ that is the tuple $(\text{cname}, \text{super}, \text{method}, \text{attr}, \text{locvar})$.

The initial state of $\Gamma$ is as follows:

$$\langle \{\text{Object, bool}\}, \{\emptyset\}, \{\emptyset\}, \{\emptyset\}, \{\emptyset\} \rangle$$

where **Object** is the super type of all classes.

When scanning the program text, $\Gamma$ is built in a stepwise way. We use primed elements to denote the states after the modification. The constructions are defined as follows.

**Class name and super class:**

```plaintext
\text{class} C_1 \text{ extends} C_2 [...], \quad C_2 \in \Gamma\.\text{name}
\Gamma'\.\text{name} = \Gamma\.\text{name} \cup \{C_1\},
\Gamma'\.\text{super} = \Gamma\.\text{super} \cup \{(C_1, C_2)\}
```

**Method:**

```plaintext
\text{class} C_1 \text{ extends} C_2 [...m(\text{param})...]
\Gamma'\.\text{method} = \Gamma\.\text{method} \cup \{(C_1, m(\text{param}))\}
```

**Constructor:**

```plaintext
\text{class} C_1 \text{ extends} C_2 [...C_1(\text{param})...]
\Gamma'\.\text{method} = \Gamma\.\text{method} \cup \{(C_1, C_1(\text{param}))\}
```

**Method Inheritance:**

```plaintext
\text{class} C_1 \text{ extends} C_2 [...], \quad (C_2, m(\text{param})) \in \Gamma\.\text{method}
\Gamma'\.\text{method} = \Gamma\.\text{method} \cup \{(C_1, m(\text{param}))\}
```

**Attribute:**

```plaintext
\text{class} C_1 \text{ extends} C_2 [...\text{pub} a : T...]
\Gamma'\.\text{attr} = \Gamma\.\text{attr} \cup \{(C_1, a : T)\}
```

where **pub** can be substitute by **prot** or **pri**.

**Attribute Inheritance (Public or Protected):**

```plaintext
\text{class} C_1 \text{ extends} C_2 [...]
\text{class} C_2 \text{ extends} [...\text{pub} a : T...]
\Gamma'\.\text{attr} = \Gamma\.\text{attr} \cup \{(C_1, a : T)\}
```

---

2 In [7] and [16], they only relate the visibility of a certain attribute to a class. As far as our concern, this is not accurate. For instance, suppose $a$ is the private attribute of class $A$, $B$ is $A$’s subclass, then $B$ can visit $a$ only via the method inherited from $A$. Hence the visibility is bound to not only classes but also methods.
Attribute Inheritance (Private):

\[
\begin{align*}
class C_1 \text{ extends } C_2 
\end{align*}
\]

\[
\begin{align*}
class C_2 \text{ extends } \ldots \{\ldots \text{prot } a : T \ldots\}
\end{align*}
\]

\[
\Gamma'. \text{attr} = \Gamma. \text{attr} \cup \{(C_1, m, a : T)\}
\]

Local Variable:

\[
\begin{align*}
(C, m, x : T) \in \Gamma. \text{method}. \text{sig}
\end{align*}
\]

\[
\Gamma'. \text{locvar} = \Gamma. \text{locvar} \cup \{(C, m, x : T)\}
\]

where \((C, m, x : T) \in \Gamma. \text{method}. \text{sig}\) stands for that \(x\) is a parameter of method \(m\) in \(C\).

Please notice that, the class can visit the inherited private attributes only via inherited methods. This is not precisely described in [7] or [16]. Another note is that we do not permit method overloading, hence there is no problem of processing order of rule “Method” and “Method Inheritance”, because if both of them exist, they will be the same.

**Type judgements.** Now we present how to get the type information of expressions in the program via \(\Gamma\).

We use type judgements \(\Gamma, C \triangleright e : T\) to mean that expression \(e\) has the type \(T\) in the scope of \(C\) under the typing environment \(\Gamma\). Sometimes the type of a local variable should be decided by a method scope. For instance, when \(x\) is a parameter of a method. In this case we use \(\Gamma, C, m \triangleright e : T\) to denote it. We have the typing rules for expressions as follows:

**true** and **false**:

\[
\Gamma, C, m \triangleright \text{true} : \text{bool}
\]

\[
\Gamma, C, m \triangleright \text{false} : \text{bool}
\]

**self**:

\[
\Gamma, C, m \triangleright \text{self} : C
\]

**null**:

\[
N \in \Gamma. \text{cname}
\]

\[
\Gamma, C, m \triangleright \text{null} : N
\]

**Attribute**:

\[
(C, m, a : T) \in \Gamma. \text{attr}
\]

\[
\Gamma, C, m \triangleright \text{self}.a : T
\]
where \( \text{visible}(C, N, x) \) stands for the variable \( x \) in class \( N \) is visible from class \( C \). It can be defined precisely following Java’s (or other language’s) convention of visibility control across classes. Because different OO languages adopt different visibility control strategies, we omit the details here for flexibility. In the remainder of this paper, sometimes we also overload the function as \( \text{visible}(C, m, x) \) or \( \text{visible}(C, m, \text{le}) \) to mean that variable \( x \) (or \( \text{le} \)) is visible in the method \( m \) within class \( C \).

Local Variable in a Method:
\[
\frac{}{\Gamma \vdash x : T}
\]

We use "\( \Gamma, C, m \vdash c \) : com" to mean that the command \( c \) is a well-formed command in the scope of \( m \) in \( C \) under the typing environment \( \Gamma \), and use \( \Gamma, C \vdash c \) : com to stand for that \( \Gamma, C, m \vdash c \) : com holds for all \( m \) in class \( C \). In the following typing rules for commands, to combine the different cases of visible attributes and local variables, we use the notation \( \text{visible}(C, m, x) \) to denote the fact that the variable \( x \) is visible in the current scope \( C, m \) where the interested command \( c \) is in.

Specification:
A specification command is well-formed if the free variables are visible in the current scope of the program.
\[
\text{FV}(\psi_i) \subseteq \alpha \quad (i = 1, 2), \quad \forall x \in \alpha \cdot \text{visible}(C, m, x)
\]
\[
\frac{}{\Gamma, C, m \vdash \alpha : [\psi_1, \psi_2] : \text{com}}
\]

Choice:
A choice command is well-formed if all its branches are well-formed and also, there is no visibility problem.
\[
\frac{}{\Gamma, C, m \vdash e_i : \text{com}, \quad (i = 1, 2), \quad \forall x \in \text{FV}(b) \cdot \text{visible}(C, m, x)}
\]
\[
\frac{}{\Gamma, C, m \vdash \text{if } b \text{ e}_1 \text{ else } e_2 : \text{com}}
\]

Sequential Composition:
A sequential composition is well-formed if all its components are well-formed.
\[
\frac{}{\Gamma, C, m \vdash e_i : \text{com}, \quad (i = 1, 2)}
\]
\[
\frac{}{\Gamma, C, m \vdash e_1 ; e_2 : \text{com}}
\]

Assignment:
A simple assignment is well-formed if its expressions are well-typed and the right hand side is a subtype of the left hand side.
\[
\frac{}{\Gamma, C, m \vdash \text{le} : C_1, \quad \Gamma, C, m \vdash e : C_2, \quad C_2 \ll C_1}
\]
\[
\frac{}{\Gamma, C, m \vdash \text{le} := e : \text{com}}
\]
Assignment by Cast:
An assignment by cast is well-formed if the corresponding expressions are well-typed and,

- The right hand side is a subtype of the left hand side.
- The casted expression is a subtype or super type of the casting type.

\[
\begin{align*}
\Gamma, C, m \triangleright le &: C_1, \\
& \quad \Gamma, C, m \triangleright e &: C_2, \\
& \quad N \ll C_1, \\
& \quad N \ll C_2 \lor C_2 \ll N \\
\Gamma, C, m \triangleright le := (N)e &: \text{com}
\end{align*}
\]

Method Invocation:\(^3\)
A method invocation is well-formed if the corresponding method does exist and the actual parameters are subtypes of formal parameters.

\[
\begin{align*}
\Gamma, N, m_0 \triangleright le &: C, \\
& \quad (C, m(\overline{para})) \in \Gamma:\text{method}, \\
& \quad \Gamma, N, m_0 \triangleright \overline{e} &: \overline{D}, \\
& \quad \overline{D} \ll \text{type}(\overline{para}) \\
\Gamma, N, m_0 \triangleright le.m(\overline{e}) &: \text{com}
\end{align*}
\]

Object Creation:
An object creation is well-formed if the corresponding expressions are well-typed and

- The actual parameters are subtypes to the ones defined in the constructor.
- The assigned value has the same type of the created object.

\[
\begin{align*}
(C, C(\overline{para})) \in \Gamma:\text{method}, \\
\Gamma, N, m \triangleright x &: C, \\
& \quad \Gamma, N, m \triangleright \overline{e} &: \overline{D}, \\
& \quad \overline{D} \ll \text{type}(\overline{para}) \\
\Gamma, N, m \triangleright C.\text{new}(x, \overline{e}) &: \text{com}
\end{align*}
\]

Object Destruction:
Any object destruction is well-formed if without any visibility problem.

\[
\begin{align*}
visible(C, m, le) \\
\Gamma, C, m \triangleright \text{free}(le) &: \text{com}
\end{align*}
\]

5.2 Method Body Lookup
In this part, we will introduce our approach of method body lookup. The system will fix the method body code to the corresponding class before the execution of the main program.

We use \(\Theta\) to denote the environment for method body lookup, which is composed of triples with the form \((C, m, \lambda \overline{x}.c)\), where \(C\) is a class name in \texttt{cname}, \(m\) is one of its methods, and \(c\) is a well-formed command. The informal meaning of the triple \((C, m, \lambda \overline{x}.c)\) is that the invocation of the method \(m\) which belongs to an object with type \(C\) is equivalent to the execution of \(c\) with a substitution of the parameters \(\overline{x}\).

\(^3\) Actually, we need another rule for \texttt{self.m()}. However, we ignore it because it is almost the same with \texttt{le.m()}. The same assumption will be adopted in the rest of this paper.
For intuitional concern, we use $\Theta, C \vdash m \rightarrow \lambda x.c$ to denote $(C, m, \lambda x.c) \in \Theta$. In the following reduction rules, one can see that in fact our approach mirrors the vtable technique in OO field. That is, each class copies all the inherited method bodies from its direct superclass. This idea is embodied by the following two rules:

The method is defined immediately:

\[
\begin{align*}
\text{class } C & \text{ extends } N \{ \ldots \{ \text{para} \} \{ c \} \}, \quad \Gamma, C, m \vdash e : \text{com} \\
\Theta, C & \vdash m \rightarrow \lambda \text{para}.c
\end{align*}
\]

Copy from immediate superclass:

\[
\begin{align*}
\text{class } C & \text{ extends } N \{ \ldots \{ undefined(C, m) \} \}, \\
\Theta, N & \vdash m \rightarrow \lambda \text{para}.c \\
\Theta, C & \vdash m \rightarrow \lambda \text{para}.c
\end{align*}
\]

where $\text{undefined}(C, m)$ stands for that the method body of $m$ in $C$ has not been filled in current state of $\Theta$.

One might argue that, to ensure that a particular super class is done before its sub-classes are, a linear order of the classes for applying the above rules in turn should be given. The answer is "yes". However, for a well typed program, the order of classes declarations in the program text is a natural order satisfying this requirement.

### 5.3 Properties

In this subsection, we will show that an expression (or command) typing does determine a derivation.

**Theorem 1.** For all typings $\Gamma, C, (m) \vdash e : T$, there is at most one derivation.

**Proof.** By induction on the structure of the expressions. Instead we can prove that, for each syntactic construct there is exactly one typing rule.

- **Case self, null, true, false.** The proof is trivial.
- **Case $x$.** We know that $x$ can only occur in the parameter list of some method. If we have $\Gamma, C, m \vdash x : T$, then it must be derived by the exact rule for local variable in a method. And plus the fact that $\Gamma, C \vdash x : T$ never holds, we can get the conclusion.
- **Case self$, a, x.a.** For each of them, the derivation approach has two forms: with $m$ or not, which depends on the precondition that holds beforehand. There is just one typing rule for each case: Attribute or Reference. By hypothesis, the conclusion holds trivially.

**Theorem 2.** For all typings $\Gamma, C, m \vdash e : \text{com}$, there is at most one derivation.

**Proof.** By **Theorem 1**, and induction on typing derivations for commands, the result holds trivially.

---

4 In fact, we need another law for constructors. But that is almost the same as this law. Thus, for simplicity, we consider the case of constructor is covered by this law. The same assumption will be adopted in the rest of this paper.
6 Weakest Precondition Semantics

In this section, we will present our main results supported by the static environments developed in the last section. The storage model and assertion language is introduced in Section 3. In this article, for the special restricted language, the local variables, attributes and \{null, self, true, false\} belong to the set Name of the OO Separation Logic.

As stated before, our language has a pure object semantics, thus, every primitive type is also object type. For simplicity, we restrict that the only primitive type here is bool. If \(x\) is a boolean variable, then \(\sigma(x) = tt\) where \(tt \in \{\text{true}, \text{false}\}\) is a reference, and \(O(\text{true}) = \text{ottrue}, O(\text{false}) = \text{offalse}\) are the only two objects of type bool, which are always in \(O\). Thus, all the values of variables in our model are references. The semantics of an expression is its value decided by the current state of the \(\sigma\) and \(O\). They are presented as follows:

\[
[\text{true}] = \text{rtrue}, \quad [\text{false}] = \text{rfalse}
\]

\[
[\text{self}]_{<\sigma,O>} = \text{self}, \quad [\text{null}]_{<\sigma,O>} = \text{rnull}
\]

\[
[\Gamma, C, m \triangleright e : T]_{<\sigma,O>} = \gamma_{<\sigma,O>} e
\]

\[
[\Gamma, C, m \triangleright e : \text{bool}] = r
\]

\[
[\Gamma, C, m \triangleright \lnot e : \text{bool}] = \lnot r
\]

where\(^5\) \(\lnot r \overset{\text{def}}{=} \begin{cases} \text{rfalse} & \text{if } r = \text{rtrue} \\ \text{rtrue} & \text{if } r = \text{rfalse} \end{cases}\)

As we have described earlier, the semantics of a command \(c\) in the model is a predicate transformer which maps any given predicate \(\psi\) to another predicate which is the weakest precondition of \(c\) w.r.t. \(\psi\). Same to [7], we define the WP semantics only for the commands which have passed the type checking, i.e., the command \(c\) satisfying \(\Gamma, C, m \triangleright c : \text{com}\). Thus the static necessities ensured by the typing environment will not appear in the preconditions defined below.

We have ordinary WP semantic definitions for specification, choice, and sequential composition.

**SPECIFICATION:**

\[
[\Gamma, N, m \triangleright \alpha : [\psi_1, \psi_2] : \text{com}]_{\psi} = \psi_1 \land (\forall \alpha : T \bullet \psi_2 \Rightarrow \psi)
\]

where \(T\) is the list of types for corresponding variables in \(\alpha\).

**CHOICE:**

\[
[\Gamma, C, m \triangleright c_1 : \text{com}]_{\psi} = f_1(\psi)(i = 1, 2)
\]

\[
[\Gamma, C, m \triangleright \text{if } b \triangleright c_1 \text{ else } c_2 : \text{com}]_{\psi} =
\]

\[
(b \Rightarrow f_1(\psi)) \land (\lnot b \Rightarrow f_2(\psi))
\]

**SEQUENTIAL COMPOSITION:**

\[
[\Gamma, C, m \triangleright c_1 : \text{com}]_{\psi} = f(\psi),
\]

\[
[\Gamma, C, m \triangleright c_2 : \text{com}]_{\psi'} = f'(\psi')
\]

\[
[\Gamma, C, m \triangleright c_1; c_2 : \text{com}]_{\psi} = f(f(\psi))
\]

\(^5\) Similar to this, we can define other logical operations such as \(\land\) and \(\lor\). Here we only give \(\lnot\) as an example.
The WP definition for assignments is defined in two cases which stand for assignment to variables and assignment to attributes respectively. Please notice that they are similar, but different to the mutation and lookup rules in [32]. The first rule stands for assignments to variables and the second one for assignments to attributes which will change the state of an object, and thus, cause the sharing (aliasing) issue.

**Assignmenent I:**

\[ \Gamma, C, m \triangleright x := e : \text{com} \psi = \psi[e/x] \]

**Assignmenent II:** (\( e \in \{x, \text{self}\} \))

\[ \Gamma, C, m \triangleright e.a := e' : \text{com} \psi = \exists o \bullet (e' \mapsto o) \wedge ((e.a \mapsto o) \wedge \psi) \]

One can compare the above two rules with the corresponding ones in traditional Separation Logic[32]:

- **Lookup (backwards reasoning):**
  \[ \exists \nu. (e \mapsto \nu) \wedge ((e \mapsto e') \mapsto p) \nu := e[p] \]

- **Mutation (backwards reasoning):**
  \[ ((e \mapsto e') \mapsto ((e \mapsto e') \mapsto p)) e := e'[p] \]

In their notation, \([e] \) stands for the content of pointer \( e \). One can see that they quantify the content \( \nu \) in the Lookup rule. However, we do not need it in the first law in our logic. The reason is that, following a pure reference semantics such as Java, in the assignment \( x := e \), \( x \) and \( e \) are both references, i.e., the members belongs to the same club. That is why we do not need to quantify anything in the first law. In contrast, we have to quantify \( o \) in the second law whereas they do not need it in the Mutation law. The reason is that \( e.a \) and \( e' \) are all references which causes the fact the \( e.a \mapsto e' \) is illegal. Thus we have to use the quantified \( o \) to denote the object referred by \( e' \).

The rules of assignment by upcast are the same as the above ones. This is not surprising — upcast does nothing assuming the assignment command has passed the type check. Also, we have two cases for variables and attributes respectively:

**Assignment By Upcast I:**

\[ \Gamma, C, m \triangleright e' : C_2, \; C_2 <: N \]

\[ \Gamma, C, m \triangleright x := (N)e : \text{com} \psi = \psi[e/x] \]

**Assignment By Upcast II:** (\( e \in \{x, \text{self}\} \))

\[ \Gamma, C, m \triangleright e.a := (N)e' : \text{com} \psi = \exists o \bullet (e' \mapsto o) \wedge ((e.a \mapsto o) \wedge \psi) \]

Following rules are for the assignment by downcast, in which we will dynamically check the type of the casted object using assertion about the object pool. Again, we have two cases:
### Definition of Assignment by Downcast I:

\[
\Gamma, C, m \triangleright e : C_2, N \lessdot C_2 \\
\exists o \bullet (\text{type}(o) \lessdot N) \land (e \mapsto o) \land ((e \mapsto o) \rightarrow \psi[e/x])
\]

### Definition of Assignment by Downcast II:

\[
\Gamma, C, m \triangleright e' : C_2, N \lessdot C_2 \\
\exists o \bullet (\text{type}(o) \lessdot N) \land (e \mapsto o) \land ((e, a \mapsto o) \rightarrow \psi)
\]

The next one is for method invocation. In this rule, we will collect all the possible method bodies of subclasses and define the weakest precondition as the disjunction of the corresponding predicates. We only give the definition for \textit{le.m}(\texttt{m}) \texttt{m}, the other case, i.e. \texttt{self.m(m)} is similar.

#### Method Invocation:

\[
\begin{align*}
\Gamma, C, m_0 \triangleright le : T, & \quad \Theta, T \triangleright m \mapsto \lambda \mathbf{para}.c, \\
\Gamma, C, m_0 \triangleright cl[le/\texttt{self}] : \texttt{com}\psi = f(\psi) \\
\exists T_1, \cdots, T_k \bullet (T_i, T) \in \Gamma.\texttt{super}, (i = 1, \cdots, k) & \quad \Theta, T_i \triangleright m \mapsto \lambda \mathbf{para}.c_i, \\
\Gamma, C, m_0 \triangleright \texttt{para} := \tau : \texttt{com}\psi = f'(\psi) \\
\Gamma, C, m_0 \triangleright c_i(T_i)le/\texttt{self} : \texttt{com}\psi = f_i(\psi)
\end{align*}
\]

\[
\exists o \bullet le \mapsto o \land (\lor_{i=0}^{k} (\text{type}(o) = T_i \land f'(f_i(\psi))))
\]

where \(T_0 \overset{\text{def}}{=} T\) and \(f_0(\psi) \overset{\text{def}}{=} f(\psi)\).

The next one is for object creation. Informally, the object creation can be imagined as the sequential composition of two "commands": the first one extends the Opool by creating a new node and returns a reference; the second one invokes the constructor to initiate the object state. That is exactly the case in OO languages practice, and in the definition here for the object creation. Please remember that we have restricted that the body of the constructor is a sequence of assignments \texttt{self.a := e}. Hence we can substitute \texttt{self} to \texttt{le} here. This is not an essential restriction, but makes the semantics simple and clear.

Because there are different forms of assignments w.r.t. the forms of \texttt{le}, i.e., \texttt{x, x.a} and \texttt{self.a}, here we have two corresponding rules for object creation. The first one creates an object and returns it to a variable while the second one returns the created object to an attribute which will cause sharing (aliasing) issue. The details are as follows.

#### Object Creation:

\[
\begin{align*}
\Theta, N \triangleright N & \mapsto \lambda \mathbf{para}.c \\
\Gamma, C, m \triangleright \texttt{para} := \tau : \texttt{com}\psi = f'(\psi) \\
\Gamma, C, m \triangleright cl[x/\texttt{self}] : \texttt{com}\psi = f(\psi)
\end{align*}
\]

\[
\forall x' : N \bullet ((x' \mapsto o_0(N)) \rightarrow f'(f(\psi))[x'/x])
\]
where \( \text{N}(\text{null}) \) is the constructor of class \( \text{N} \) and \( \text{o}_0(\text{N}) \) stands for the object with type \( \text{N} \) in which all the attributes’ initial value is \( \text{rnull} \).

Different languages adopt different attitudes to the object destruction. For examples, C++ offers an operator for explicitly deleting an object while Java leaves the task to the garbage collector. Here, as an alternative rule, we offer the semantics of object destruction isolated.

**OBJECT DESTRUCTION:**

\[
\begin{align*}
[\Gamma, C, m \triangleright \text{null} : \text{com}] \psi &= f(\psi) \\
[\Gamma, C, m \triangleright \text{free(} \text{le} \text{)} : \text{com}] \psi &= (\text{le} \mapsto -) * f(\psi)
\end{align*}
\]

If the language, such as C++, offers the mechanism of *destructor*, denoted by \( \sim \text{C}() \), we can further modify the above rule to a new form as follows:

\[
\begin{align*}
[\Gamma, C, m \triangleright \text{le.} (\sim \text{C}(\text{}))] \psi &= g(\psi) \\
[\Gamma, C, m \triangleright \text{null} : \text{com}] \psi &= f(\psi) \\
[\Gamma, C, m \triangleright \text{free(} \text{le} \text{)} : \text{com}] \psi &= g((\text{le} \mapsto -) * f(\psi))
\end{align*}
\]

where, as one can expect, \( \sim \text{C}(\text{)} \) is the destructor of class \( \text{C} \).

### 6.1 Properties

This subsection shows that the semantics is a well defined function on well defined commands.

Before proving the main theorem, we introduce a lemma first.

**Lemma 1.** Suppose \( \Gamma, C, m \triangleright c : \text{com} \). The free variables in the command \( c \) are denoted by \( \mathcal{T} \) (including the particular \text{self}) and their types by \( \text{type}(\mathcal{T}) \). If \( \mathcal{T} \prec: \text{type}(\mathcal{T}) \), and \( \Gamma, D, m' \triangleright e : \mathcal{T} \) is derivable, then \( \Gamma, D, m' \triangleright c[\mathcal{T}/\mathcal{T}] \) is derivable.

**Proof.** By structural induction on commands \( c \). \( \square \)

**Theorem 3.** Suppose we have built the environment \((\Theta, \Gamma)\). For all derivable command \( \Gamma, C, m \triangleright c : \text{com} \), the semantics \([\Gamma, C, m \triangleright c : \text{com}]\) is a total function on all formulas, regardless of type.

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

- **Case** Specification statement: For this case, there is a single typing rule, and the semantics is given directly.
- **Case** Choice, Sequential composition, Object destruction: In each case, there is a direct semantic definition and the conclusion holds by induction hypothesis.
- **Case** Assignment: There are two semantic rules for assignment depending on the forms of \( \text{le} \). From the syntactic definition, \( \text{le} \) has three forms: \( x, x.a, \text{self}.a \), and correspondingly, for each form, the direct semantic rule is given uniquely. So the conclusion holds.
- **Case** Assignment by cast: The semantics is given in two ways: one for upcast, and the other for downcast. And at the same time, from the typing rule for assignment by cast, we can see that it exactly includes these two kinds: upcast and downcast. For upcast and downcast, the semantic rules are given in two forms in term of the forms of \( le \), same as in simple assignment. The conclusion holds similarly.

- **Case** Method invocation: Let \( \psi \) be any formula. The semantics for \( le.m(\tau) \) is defined provided that, \( f(\psi), f'(\psi) \) and for every \( i = 1, \ldots, k, f_i(\psi) \) are defined for \( \psi \). First, because \( le.m(\tau) \) is well typed, then from the typing rules, we have \( (T, m(\text{para})) \in \Gamma \). By Lemma 5, \( \psi \) for \( \tau \) in the syntax of Section 3, then \( [\Gamma, C, m \triangleright e/\text{self}] : \text{com} \). By Lemma 1, we can get that \( \Gamma, C, m_0 \triangleright c[\psi/\text{para}] : \text{com} \). By induction hypothesis, \( f(\psi) \) exists for some \( \psi \); and also from \( \text{type}(\tau) <: \text{type}(\text{para}) \), we can get the fact \( \Gamma, C, m_0 \triangleright \text{para} := \text{self} : \text{com} \). By induction hypothesis, \( f(\psi) \) is defined. Second, we are going to give the existence of \( f_i(\psi) \) for each \( i = 1, \ldots, k \). Note that in the substitution of \( c_i \), we use \( (T_i)le/\text{self} \). From the premise that \( \Theta, T_i \triangleright m \triangleright \lambda \text{para}.c_i \), and by Lemma 1, we can get that \( \Gamma, C, m \triangleright c_i[(T_i)le/\text{self}] : \text{com} \). Then by induction hypothesis, \( f_i(\psi) \) exists as above. So the conclusion holds.

- **Case** Object Creation: From the definition of semantics for Object Creation, we can see that if for some \( \psi \), there exists \( f(\psi), f'(\psi) \), then the conclusion holds. Same to the case for method invocation, because \( \Gamma, C, m \triangleright N.x.(x, \tau) : \text{com} \), we have that the class \( N \) has a well formed constructor, which leads to the fact that there exists \( c \) which satisfies that \( \Theta, N \triangleright N \triangleright \lambda \text{para}.c \), and also \( \Gamma, N \triangleright c : \text{com} \). And plus the fact that \( \Gamma, C, m \triangleright x : N \), by Lemma 1, we can get the conclusion \( \Gamma, C, m \triangleright c[x/\text{self}] : \text{com} \). And also we have \( \text{type}(\tau) <: \text{type}(\text{para}) \), so \( \Gamma, C, m \triangleright \text{para} := \text{self} : \text{com} \). So by induction hypothesis, the conclusion holds.

**Theorem 4.** Suppose we have built \((\Theta, \Gamma)\). For all derivable command \( \Gamma, C, m \triangleright c : \text{com} \), if \( \psi \) is a well formed predicate defined in the syntax of Section 3, then \([\Gamma, C, m \triangleright c : \text{com}])\psi \) is also a well formed predicate defined in that syntax.

**Proof.** Straightforward induction on the structure of commands. \( \square \)

**Theorem 5.** Suppose \( f : \Psi \rightarrow \Psi \) is any defined WP predicate transformer and \( \psi', \psi'' \) are any well formed predicates. If we have \( \psi' \Rightarrow \psi'' \), then \( f(\psi') \Rightarrow f(\psi'') \).

**Proof.** By Theorem 4, \( f(\psi) \) exists for every well formed predicate \( \psi \) and all derivable command \( c \). Now we prove the conclusion by structural induction on command \( c \).

- **Case** Specification: \( f(\psi) = \psi_1 \land (\forall \alpha : T \bullet \psi_2 \Rightarrow \psi) \).
  If \( \psi' \Rightarrow \psi'' \), then from \( \forall \alpha : T \bullet \psi_2 \Rightarrow \psi' \), we can get \( \forall \alpha : T \bullet \psi_2 \Rightarrow \psi'' \) by transition of implication, which makes the conclusion hold.

- **Cases** Choice, Sequential Composition: By induction hypothesis, we can get the conclusion holds for each case.

- **Case** Assignment I, Assignment by Upcast I, Assignment by Downcast I: We need to prove that if \( \psi' \Rightarrow \psi'' \), then \( \psi'[e/x] \Rightarrow \psi''[e/x] \). Take \( \psi' \Rightarrow \psi'' \) as a new predicate. The fact is trivial that the substitution of the variables in predicate doesn’t influence its value. So the conclusion holds.
- **Cases** Assignment II, Assignment by Upcast II, Assignment by Downcast II: By the definition of the separating conjunction and separating implication, the conclusion holds easily.

- **Case** Method Invocation: If \( \psi' \Rightarrow \psi'' \), then by induction hypothesis, we can get that for each \( i = 0, 1, \ldots, k \), \( f_i(\psi') \Rightarrow f_i(\psi'') \) and \( f'(\psi') \Rightarrow f'(\psi'') \). Then by the property of the operators \( \land, \lor \), the conclusion holds.

The Cases for Object Creation, Object Destruction are also the same as above. From the definition of the assertion operators and by induction hypothesis, we can get the conclusion very easily.

### 6.2 Examples

**Example for reference types.**

Suppose we have \( C_1 <: C_2 <: C_3 \), and \( e \) is a parameter suitable to the constructor of \( C_1 \). Let us consider the following class:

```plaintext
Class A extends Object{
    pri a1 : C1, a2 : C2, a3 : C3;
    m()
        C1.new(self.a1, e);
        self.a3 := (C3)self.a1;
        self.a2 := (C2)self.a3
    }
```

One can see that from **THEOREM 3**, we have the above method \( m \) has a well defined semantics. But in [7], due to their absence of object sharing, the semantics of downcast assignment

\[
\text{self.a2} := (C2)\text{self.a3}
\]

is not defined.

**Example for method invocation.**

Suppose we have the following class declarations:

```plaintext
Class A extends Object{
    prot a, b : T;
    m(x : T)
        x := self.a;
    }
}

Class B extends A{
    m(x : T)
        x := self.b;
    }
```
Class $C$ extends Object{
    prot $r : A$;
    prot $s : B$;
    ...
    ...
}

According to our rules for method invocation and assignment, we have the following results:

\[
\begin{align*}
[[\Gamma, C \triangleright s.m(x) : \text{com}] \psi &= \exists o \bullet (s.a \mapsto o) \ast ((s.a \mapsto o) \rightarrow \psi[s.a/x]) \\
[[\Gamma, C \triangleright r.m(x) : \text{com}] \psi &= \exists o \bullet (r.a \mapsto o) \ast ((r.a \mapsto o) \rightarrow \psi[r.a/x]) \lor \\
&\exists o \bullet (r.b \mapsto o) \ast ((r.b \mapsto o) \rightarrow \psi[r.b/x])
\end{align*}
\]

7 Soundness and Completeness

After presenting the WP semantics, we prove the soundness and completeness of the semantics in this section. We will give an operational semantics on our storage model first. And then discuss the soundness and completeness of the WP semantics with respect of that.

7.1 Operational Semantics

In OO Separation Logic model, expressions are interpreted depending on both the store and the Opool. The semantics of an expression $e$ is a reference value, denoted by $[e]\sigma O$. For simplicity, ignoring $\sigma$ and $O$, we only write $[e]$ when there is no confusion throughout this paper. The definitions are the same as the ones defined in the WP semantics. In the following we will give the definition for commands.

The commands in the OO language are interpreted by giving the operational semantics in our storage model. We define a transition relation $\rightarrow$ over configurations. The configurations have two components:

1. a non-terminated command-state pair $(c, (\sigma, O))$, in which $\text{FV}(c) \subseteq \text{dom}(\sigma)$ for some $\sigma \in \text{store}$, $O \in \text{Opool}$.
2. a terminated state $(\sigma, O)$ for some $\sigma \in \text{store}$, $O \in \text{Opool}$, which states that the execution of some command has completed.

For some well typed expression $e$, we can get its type from the static environment $\Gamma$ according to the typing rules in Section 6, and also can get the type of the object it refers to as described in Section 2. Here for simplicity, we assume that both the two types can be acquired directly, denoted by $\text{type}(e)$ and $\text{type}(O(\gamma < \sigma, O > e))$ respectively.
In this semantics, the commands are specified by the following inference rules.

The operational semantics for assignment is defined in two cases: One is assignment to variables, the execution of which leads to the change of the store \( \sigma \); while the other one is assignment to attributes. It results in a new Opool with the store remaining unchanged.

**Assignment I:**

\[
\begin{align*}
\gamma_{<\sigma,O}>e &\in \text{dom}(O) \\
\langle x := e, (\sigma, O) \rangle \rightsquigarrow ([\sigma|x : \gamma_{<\sigma,O}>e], O)
\end{align*}
\]

**Assignment II:** \( e \in \{x, \text{self}\} \)

\[
\begin{align*}
\gamma_{<\sigma,O}>e &\neq \text{null}, \ \gamma_{<\sigma,O}>e' \in \text{dom}(O), \\
O(\gamma_{<\sigma,O}>e) = o, \ a \in O(\gamma_{<\sigma,O}>e)_1 \\
\langle e.a := e', (\sigma, O) \rangle \rightsquigarrow (\sigma, [O|\gamma_{<\sigma,O}>e \mapsto (o|a : \gamma_{<\sigma,O}>e')])
\end{align*}
\]

The operational semantics for assignment by cast are illustrated in two cases: upcast and downcast. The first one is similar to assignment; while the second one depending on the dynamic type of the casted object. Both of them will be defined in term of assignment to variables or attributes.

**Assignment by Upcast I:**

\[
\begin{align*}
\gamma_{<\sigma,O}>e &\in \text{dom}(O), \ \text{type}(e) <: N \\
\langle x := (N)e, (\sigma, O) \rangle \rightsquigarrow ([\sigma|x : \gamma_{<\sigma,O}>e], O)
\end{align*}
\]

**Assignment by Upcast II:** \( e \in \{x, \text{self}\} \)

\[
\begin{align*}
\gamma_{<\sigma,O}>e &\neq \text{null}, \ O(\gamma_{<\sigma,O}>e) = o, \ a \in O(\gamma_{<\sigma,O}>e)_1 \\
\gamma_{<\sigma,O}>e' \in \text{dom}(O), \ \text{type}(e') <: N \\
\langle e.a := (N)e', (\sigma, O) \rangle \rightsquigarrow (\sigma, [O|\gamma_{<\sigma,O}>e \mapsto (o|a : \gamma_{<\sigma,O}>e')])
\end{align*}
\]

**Assignment by Downcast I:**

\[
\begin{align*}
\gamma_{<\sigma,O}>e &\in \text{dom}(O), \ O(\gamma_{<\sigma,O}>e) = o, \\
\text{type}(o) <: N <: \text{type}(e) \\
\langle x := (N)e, (\sigma, O) \rangle \rightsquigarrow ([\sigma|x : \gamma_{<\sigma,O}>e], O)
\end{align*}
\]

**Assignment by Downcast II:** \( e \in \{x, \text{self}\} \)

\[
\begin{align*}
\gamma_{<\sigma,O}>e &\neq \text{null}, \ O(\gamma_{<\sigma,O}>e) = o_1, \ a \in O(\gamma_{<\sigma,O}>e)_1 \\
O(\gamma_{<\sigma,O}>e') = o_2, \ \text{type}(o_2) <: N <: \text{type}(e') \\
\langle e.a := (N)e', (\sigma, O) \rangle \rightsquigarrow (\sigma, [O|\gamma_{<\sigma,O}>e \mapsto (o_1|a : \gamma_{<\sigma,O}>e')])
\end{align*}
\]

The execution of Method Invocation captures the dynamic binding feature. As seen in the operational rule, the correct method body is guaranteed to be executed according to the dynamic object type of the invocator \( le \).

**Method Invocation:**

\[
\begin{align*}
\text{type}(O(\gamma_{<\sigma,O}>e)) = T', \ \Theta, T' \triangleright m &\rightsquigarrow \lambda\text{para}.e \\
\langle \text{para} := \bar{e}; e(T')|l/\text{self}|_1, (\sigma, O) \rangle \rightsquigarrow (\sigma', O') \\
\langle le.m(\bar{e}), (\sigma, O) \rangle \rightsquigarrow (\sigma', O')
\end{align*}
\]
In the following two operational rules for object creation, \( r \) is chosen as a fresh reference which is not contained in the domain of the current Opool \( O \). We can see that the selection of \( r \) is non-deterministic. The execution of \( N.\text{new}(le, \vec{r}) \) first creates an object of type \( N \) with \( \text{rnul} \) as values of all its attributes, and then choose a new reference \( r \) to refer to it. Followingly, \( le \) will refer to \( r \), and execute the constructor of class \( N \) to initiate \( le \) with parameters \( \vec{r} \).

**Object Creation:**
For some \( r \) not in \( \text{dom} \( O \) \), suppose we have \( \sigma' \overset{\text{def}}{=} |\sigma[x : r]| \), and \( O^r \overset{\text{def}}{=} [O|r \mapsto (\text{attr}(N) : \text{rnul})] \). Then

\[
\begin{aligned}
\Theta, N \triangleright \llbracket \sigma \racket, \\
\llbracket N.\text{new}(x, \vec{r}), \langle \sigma, O \rangle \rrbracket \rightsquigarrow \langle \sigma'', O'' \rangle \\
\end{aligned}
\]

In object destruction, the execution of \( \text{free}(le) \) will delete the corresponding object that \( le \) refers to from the current Opool, and then let \( le \) refers to no object.

**Object Destruction:**
Suppose we have \( \sigma' \overset{\text{def}}{=} \sigma \), and \( O'' \overset{\text{def}}{=} O' \). Then

\[
\begin{aligned}
\gamma_{<\sigma, O> \triangleright le} \in \text{dom}(O), \\
\llbracket \langle le := \text{null}, (\sigma', O') \rangle \rrbracket \rightsquigarrow \langle \sigma'', O'' \rangle \\
\end{aligned}
\]

As in Section 6, we have an alternative operational rule for object destruction, which uses the destructor mechanism to do the destruction as adopted in C++.

Suppose we have \( \sigma'' \overset{\text{def}}{=} \sigma' \), and \( O'' \overset{\text{def}}{=} O' \). Then

\[
\begin{aligned}
\langle le.\text{(C())}, (\sigma, O) \rangle \rightsquigarrow \langle \sigma', O' \rangle \\
\langle le := \text{null}, (\sigma'', O'') \rangle \rightsquigarrow \langle \sigma'', O'' \rangle \\
\end{aligned}
\]

The operational rules for Choice and Sequential Composition are ordinarily defined.

**Choice:**

\[
\begin{aligned}
\gamma_{<\sigma, O> \triangleright b = \text{true}} \quad \langle c_1, (\sigma, O) \rangle \rightsquigarrow \langle \sigma', O' \rangle \\
\langle \text{if} \ b \ c_1 \ \text{else} \ c_2, (\sigma, O) \rangle \rightsquigarrow \langle \sigma', O' \rangle \\
\end{aligned}
\]

**Sequential Composition:**

\[
\begin{aligned}
\langle c_1, (\sigma, O) \rangle \rightsquigarrow \langle \sigma', O' \rangle, \\
\langle c_2, (\sigma', O') \rangle \rightsquigarrow \langle \sigma'', O'' \rangle \\
\langle c_1; c_2, (\sigma, O) \rangle \rightsquigarrow \langle \sigma'', O'' \rangle \\
\end{aligned}
\]

Having given the operational semantic rules for all commands, we have the following notations. We say that the configuration \( \langle c, (\sigma, O) \rangle \) gets stuck, if there exists no configuration \( A \) such that \( \langle c, (\sigma, O) \rangle \rightsquigarrow A \) according to the operational rules. For example, the users may attempt to access or delete a non-existed object. In contrast to this, we define \( \langle c, (\sigma, O) \rangle \) is safe if there exists a terminated configuration \( \langle \sigma', O' \rangle \) such that \( \langle c, (\sigma, O) \rangle \rightsquigarrow^* \langle \sigma', O' \rangle \).
7.2 Soundness

In this subsection, we prove the soundness of the WP semantics. Informally, a WP semantics is sound, if the following statement holds for every command: From any state satisfying the defined weakest precondition $\psi^0$ w.r.t. the command $c$, any possible final state after the execution of $c$ satisfies the postcondition $\psi$. It is formally defined as follows and proved immediately.

**Definition 1 (Soundness).** Suppose $\Psi$ is the space of legal predicates and COM is the space of legal commands. For any given WP predicate transformer $[\cdot] : \Psi \times \text{COM} \rightarrow \Psi$, we define it is sound if for any predicates $\psi, \psi^0 \in \Psi$ and command $c \in \text{COM}$ satisfying $[I, C', m \triangleright c : \text{com}] \psi = \psi'$, we have:

$$\forall (\sigma, O) \bullet [\psi^0](\sigma, O) \implies (\forall (\sigma', O') \bullet (\text{if } (c, (\sigma, O)) \leadsto^* (\sigma', O') \text{ then } [\psi](\sigma', O')))$$

**Theorem 6 (Soundness).** The weakest precondition semantics for commands defined in Section 6 is sound.

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

- **Case Assignment I, “$x := e$”**

  Assume that $\sigma, O$ satisfies the precondition, i.e., $[\psi[e/x]](\sigma, O)$. By the operational semantics of Assignment I, we have that

  $$\langle x := e, (\sigma, O) \rangle \leadsto^* (\sigma[x : \gamma_{<\sigma, O}e], O)$$

  Hence we need to show that $[\psi](\sigma', O)$ holds, in which $\sigma' = [\sigma[x : \gamma_{<\sigma, O}e]]$. Because we have $[\psi[e/x]](\sigma, O)$, and the assignment for variable does not cause any side effect such as aliasing. So we can conclude that after the assignment, $[\psi](\sigma', O)$ holds.

- **Case Assignment II, “$e.a := e'$”**

  This is an important case in our semantics because it capture the aliasing problem. Assume that $\sigma, O$ satisfies the precondition, i.e., $[\exists a \bullet (e' \leadsto a) \star ((e.a \leadsto a) \star e)](\sigma, O)$.

  By the operational semantics for Assignment II, we have that

  $$\langle e.a := e', (\sigma, O) \rangle \leadsto (\sigma, [O][\gamma_{<\sigma, O}e \leadsto o(a : \gamma_{<\sigma, O}e')])$$

  Again we need to show $[\psi](\sigma, O')$ holds, in which $O' = [O][\gamma_{<\sigma, O}e \leadsto o(a : \gamma_{<\sigma, O}e')]$. The assertion $e' \leadsto o$ in the precondition ensures that $\gamma_{<\sigma, O}e' \in \text{dom}(O)$ and does not points to a null object, so $e.a$ is legal and the assignment does not get stuck.

  From the assumption and the definition of $\star$ and $\leadsto$, we have that if the object pool $O'$ satisfies the fact that the object $o$ referred by $e'$ is also referred by $e.a$, then we have $[\psi](\sigma, O')$. This is exactly the result of the operational rule. Thus we obtain that $[\psi](\sigma, O')$ holds.
– **Case** Assignment by Upcast, I: “\(x := (N)e\)” and II, “\(e.a := (N)e'\)”

Because the assignment by upcast has the same effect of the simple assignment, the proofs for these two cases are almost the same with the proofs for the cases of Assignment I and II. What differs is that the premise of the semantics. It can be noticed that \(C_2 \prec N\) holds in the premise of the weakest precondition semantics. It can imply the premise of the operational transition that \(type(e) \prec N\). Thus, we can complete the proofs in the same way.

– **Case** Assignment by Downcast I, “\(x := (N)e\)”

Assume that \(\sigma, O\) satisfies the precondition, i.e., \(\{\exists o \bullet (e \mapsto o) \ast ((e \mapsto a) = *\psi)\}[e/x]\)(\(\sigma, O\)).

By the operational semantics for Assignment by Downcast I, we have that

\[
(x := e, (\sigma, O)) \leadsto ([\sigma[x : \gamma_{<\sigma,O}e]], O)
\]

Hence we need to show that \(\psi(\sigma', O)\) holds, in which \(\sigma' = [\sigma[x : \gamma_{<\sigma,O}e]].\)

The assertion \(e \mapsto o\) in the precondition ensures that \(\gamma_{<\sigma,O}e \in \text{dom}(O)\), so the assignment does not get stuck. And \(type(o) \prec N\) ensures that the cast can be correct performed.

Please notice that \(N \prec C_2\) holds in the premise of the weakest precondition semantics. It can imply the premise of the operational transition that \(type(o) \prec N \prec type(e)\).

From the assumption, and the definitions of \(\ast\), we have \(\psi[e/x]\) holds if the object pool \(O\) satisfies the fact that \(e\) refers a certain object with suit type. So after the assignment we have that \(\psi(\sigma', O)\).

– **Case** Assignment by Downcast II, “\(e.a := (N)e'\)”

Assume that \(\sigma, O\) satisfies the precondition, i.e., \(\{\exists o \bullet (e \mapsto a) \ast ((e,a \mapsto a) = *\psi)\}[e/x]\)(\(\sigma, O\)).

By the operational semantics for Assignment by Downcast II, we have that

\[
(e.a := e', (\sigma, O)) \leadsto (\sigma, [O[\gamma_{<\sigma,O}e \mapsto (o[a : \gamma_{<\sigma,O}e'])])
\]

Hence we need to show that \(\psi(\sigma, O')\) holds, in which \(O' = [O[\gamma_{<\sigma,O}e \mapsto (o[a : \gamma_{<\sigma,O}e'])]\).

The assertion \(e \mapsto o\) in the precondition ensures that \(\gamma_{<\sigma,O}e \in \text{dom}(O)\) and does not points to a null object, so \(e.a\) is legal and the assignment does not get stuck. And \(type(o) \prec N\) ensures that the cast can be correct performed.

Please notice that \(N \prec C_2\) holds in the premise of the weakest precondition semantics. It can imply the premise of the operational transition that \(type(o_2) \prec N \prec type(e')\).

From the assumption, and the definition of \(\ast\) and \(\ast\) we have if the object pool \(O\) satisfies the condition, so we can conclude that \(\psi(\sigma, O')\) holds.

– **Case** Method invocation: “\(e = le.m(\pi)\)”
Suppose we have
\[ \{ \psi \}(\sigma, O) \]
and
\[ \langle le.m(\overline{x}), (\sigma, O) \rangle \rightarrow (\sigma', O') \]
where \( \psi' = \exists o \cdot le \leftarrow o \land (\forall_i^k \text{type}(o) = T_i \land f'(f_i(\psi))) \) is the weakest precondition for the formula \( \psi \) defined in this paper. What we have to prove is that \( \{ \psi \}(\sigma', O') \).

From the fact that \( \{ \psi' \}(\sigma, O) \), we can get that for some \( i \), \( \exists o \cdot le \leftarrow o \land (\forall_i^k \text{type}(o) = T_i \land f'(f_i(\psi))) \). So from the weakest semantics for method invocation, we can get that the fact \( \Gamma, C, m_0 \triangleright m_1 := \overline{x} : \text{com} \psi = f'(\psi) \) and \( \Gamma, C, m_0 \triangleright c_1[(\overline{x}/e)/\text{self}] : \text{com} \psi = f_i(\psi) \) holds, and also the fact that \( T_i = T', c_i = c \), in which \( T', c \) is as in the definition of the operational semantics for method invocation. So from the fact that \( \{ f_i(\psi) \}(\sigma, O) \), we have the conclusion \( \{ \psi \}(\sigma', O') \) by induction hypothesis.

- **Case Object Creation.** “\texttt{N.new}(x, \overline{x})”

Suppose we have
\[ \langle \texttt{N.new}(x, \overline{x}), (\sigma, O) \rangle \rightarrow (\sigma'', O'') \]
and
\[ \{ \psi \}(\sigma, O) \]
where \( \psi' = \forall x' : N \bullet ((x' \rightarrow o_0(N)) \rightarrow f'(f(\psi))[x'/x]) \) is the weakest precondition defined in this paper.

What we need to show is that \( \{ \psi \}(\sigma'', O'') \).

We can consider the execution of object construction as two steps: The first one create a new object \( o_0(N) \) and assign it to \( x \), the second one initiate \( o_0(N) \) by executing the commands in the constructor.

For \( \forall x' \), take \( \sigma^0 \overset{\text{def}}{=} [\sigma[x' : r], \text{and } O^0 \overset{\text{def}}{=} \Gamma_0 \triangleright \text{attr}(N) : \text{null}] \). Because we have \( \{ \psi \}(\sigma, O) \), by the definition of \( \ast \), we have that \( \{ f'(f(\psi))[x'/x] \}(\sigma^0, O^0) \). Let \( \sigma' \overset{\text{def}}{=} [\sigma[x : r], \text{and } O' \overset{\text{def}}{=} \Gamma_0 \triangleright \text{attr}(N) : \text{null}] \), we can get the fact \( \{ f'(f(\psi))[x'/x] \}(\sigma', O') \).

Because we have \( \Gamma_0 \triangleright \overline{	ext{attr}}[x/\text{self}] : \text{com} \psi = f'(f(\psi)) \) in the premise, in conjunction of the induction hypothesis, we have that \( \{ \psi \}(\sigma'', O'') \) holds.

- **Case Object Destruction.** “\texttt{free}(le)”

Here we only prove the first rule of weakest precondition semantics for object destruction is sound, while the idea of proving the other one is similar.

Assume the precondition holds for \( (\sigma, O) \), i.e., \( \{ le \rightarrow \_ \} \ast f(\psi) \)(\sigma, O) and \( \{ \texttt{free}(le), (\sigma, O) \} \rightarrow (\sigma'', O'') \). What we have to prove is that \( \{ f(\psi) \ast \}_o \) holds. We can know that \( le \) refers to some object. From the definition of \( \ast \), we can know that \( \{ f(\psi) \}(\sigma', O') \), where \( O' = O \setminus \text{dom}(O) \ast \langle \sigma, O, le \rangle \). And further, because \( \sigma' = \sigma \) defined in the operational semantics for object destruction, \( [f(\psi)](\sigma', O') \) holds. Because \( f(\psi) \) is also the precondition of the command \( le := \text{null} \) for \( \psi \),
and \( \langle e := \text{null}, (\sigma', O') \rangle \rightarrow (\sigma'', O'') \), by the induction hypothesis, we can get
\[ [\psi](\sigma'', O'') \], which makes the conclusion hold.

The proofs for structural commands are similar to the ones in ordinary imperative languages. Here we list one of them formally.

- **Case Sequential Composition, \( \langle c_1;c_2 \rangle \)**

  Assume \( (\sigma, O) \) satisfies the precondition, i.e., \( \{f(f'(\psi))\}(\sigma, O) \) and \( \langle c_1;c_2, (\sigma, O) \rangle \rightarrow (\sigma'', O'') \). What we have to prove is that \( [\psi](\sigma'', O'') \). From the weakest precondition semantics for sequential, we can also consider \( f(f'(\psi)) \) as the precondition of \( c_1 \) for the predicate \( f'(\psi) \). According to the operational semantics for sequential, by the induction hypothesis, we get the fact \( [f'(\psi)](\sigma', O') \). Because \( f'(\psi) \) is the precondition of \( c_2 \) for \( \psi \), and also \( \langle c_2, (\sigma', O') \rangle \rightarrow (\sigma'', O'') \), we can get the conclusion \( \psi(\sigma'', O'') \) by the induction hypothesis.

### 7.3 Completeness

In this subsection, we prove the completeness of the WP semantics. Informally, a WP semantics is complete, if the WP semantics really gives the weakest precondition, i.e. any other precondition can imply the defined one. In other words, we have the following equivalent description:

For every command \( c \), we have that if from a state which can reach a state satisfying the postcondition \( \psi \) after the execution of \( c \), then the weakest precondition \( \psi \) defined w.r.t. the command \( c \) should holds on the state. It is formally defined as follows and proved immediately.

**Definition 2 (Completeness).** For any given WP predicate transformer \( \llbracket - \rrbracket : \Psi \times \text{COM} \rightarrow \Psi \), it is complete if for any predicates \( \psi, \psi' \in \Psi \) and command \( c \in \text{COM} \) satisfying \( \llbracket \Gamma; C, m \triangleright c : \text{com} \rrbracket \psi = \psi' \), we have:

\[
\forall (\sigma', O') \bullet [\psi](\sigma', O') \implies [\psi](\sigma, O) \bullet \text{if } \langle (c, (\sigma, O)) \rangle \rightarrow^* (\sigma', O') \text{ then } [\psi](\sigma, O))
\]

**Theorem 7 (Completeness).** The weakest precondition semantics for commands defined in Section. 6 is complete.

**Proof** Again, the proof is given by induction on the structure of commands:

- **Case Assignment I, \( \langle x := e \rangle \)**

  Suppose we have

  \[
  \langle x := e, (\sigma, O) \rangle \rightarrow ([\sigma|x : \gamma_{<\sigma,O}>e], O)
  \]

  and \( [\psi](\sigma|x : \gamma_{<\sigma,O}>e), O) \). We need to show that \( [\psi[e/x]](\sigma, O) \) holds. Because \( \psi \) holds after the assignment, and this assignment does not cause any sharing problem, so the effect of the assignment is substituting \( x \) with \( e \) only, which implies \( [\psi[e/x]](\sigma, O) \). Thus we obtain the conclusion.
– **Case** Assignment II, “\(e.a := e'\)”

Suppose we have

\[
\{ e.a := e', (\sigma, O) \} \leadsto ((\sigma, [O]_{\gamma<\sigma,O>e} e \mapsto (\sigma|a : \gamma<\sigma,O>e')))
\]

And \(\{\psi\}(\sigma, O')\), where \(O' = [O]_{\gamma<\sigma,O>e} e \mapsto (\sigma|a : \gamma<\sigma,O>e')\).

What we need to show is that \(\{\psi'\}(\sigma, O)\), where \(\psi' = \exists o \bullet (e' \mapsto o) \ast ((e.a \mapsto o) \ast \psi)\) is the weakest precondition defined in this paper.

Take \(o\) defined as \(O(\gamma<\sigma,O>e')\), immediately we have \(\exists O_o \subseteq O \bullet ((\sigma, O_0)\).

Furthermore, if \(e.a\) also refers to \(o\), then the true assertions on \(O'\) should hold.

By the definitions of \(\ast\) and \(\ast\), we have \((e' \mapsto o) \ast ((e.a \mapsto o) \ast \psi)\) holds. i.e. \(\{\psi'\}(\sigma, O)\).

– **Case** Assignment by Upcast, I: “\(x := (N)e\)” and II, “\(e.a := (N)e'\)”

Because the assignment by upcast has the same effect of the simple assignment, the proofs for these two cases are almost the same with the proofs for the cases of Assignment I and II. What differs is that the premise of the semantics.

It can be noticed that that because the premise of the operational transition, we have that \(type(e) < N\), which can imply that the premise of the weakest precondition semantics \(C_2 < N\) also holds. Thus, we can complete the proofs in the same way.

– **Case** Assignment by Downcast I, “\(x := (N)e\)”

Suppose we have

\[
\{ x := (N)e, (\sigma, O) \} \leadsto ([\sigma|x : \gamma<\sigma,O>e], O)
\]

and \(\{\psi\}([\sigma|x : \gamma<\sigma,O>e], O)\). What we need to show is that \(\{\psi'\}(\sigma, O)\), where \(\psi' = \exists o \bullet (type(o) < N) \land (e \mapsto o) \ast ((e \mapsto o) \ast \psi[e/x])\).

Please notice that because the premise of the operational transition, we have that \(type(o) < N < type(e)\), which can imply that the premise of the weakest precondition semantics \(C_2 < N\) also holds.

Because \(\{\psi\}([\sigma|x : \gamma<\sigma,O>e], O)\), we have \(\{\psi[e/x]\}(\sigma, O)\). According to the premise of the transition rule, we have \(\gamma<\sigma,O>e \in \text{dom}(O)\). Thus we can take \(o = O(\gamma<\sigma,O>e)\) and conclude that

\[
\exists o \bullet (e \mapsto o) \ast (\psi[e/x])
\]

By the property of \(\ast\), the following assertion is weaker than the above one:

\[
\exists o \bullet (e \mapsto o) \ast ((e \mapsto o) \ast \psi[e/x])
\]

Also, the premise \(type(o) < N < type(e)\) ensures that \(\{\sigma, O\}\) holds. By combining the above two assertions, we finally reach that \(\{\sigma, O\}\psi'\). Thus we obtain the conclusion.

– **Case** Assignment by Downcast II, “\(e.a := (N)e'\)”
Suppose we have

\[(e.a := e', (\sigma, O)) \leadsto (\sigma, [O][\gamma<\sigma, O>e \mapsto (a : \gamma<\sigma, O>e')])\]

And \(\{\psi\}(\sigma, O')\), where \(O' = [O][\gamma<\sigma, O>e \mapsto (a : \gamma<\sigma, O>e')]\)

What we need to show is that \(\{\psi'\}(\sigma, O)\) where \(\psi' = \exists o \bullet (\text{type}(o)) \land (e \mapsto a) \cdot ((e.a \mapsto o) \Rightarrow \psi)\) is the weakest precondition defined in this paper.

Please notice that because the premise of the operational transition, we have that \(\text{type}(o_2) \land N < \text{type}(e')\), which can imply that the premise of the weakest precondition semantics \(N < C_2\) also holds.

Take \(o = O(\gamma<\sigma, O>e')\), immediately we have \(\exists O_o \subseteq O \bullet (e' \mapsto o, O_o)\).

Furthermore, if \(e.a\) also refers to \(o\), then the true assertions on \(O'\) should hold. By the definitions of \(*\) and \(+\), we have \((e' \mapsto o) \ast ((e.a \mapsto o) \Rightarrow \psi)\) holds.

Also, the premise \(\text{type}(o_2) \land N < \text{type}(e')\) ensures \(\{\text{type}(o) \land N\}(\sigma, O')\). By combining the above assertions, we finally reach that \(\{\psi'\}(\sigma, O)\).

**Case Method invocation: “c = le.m(\tau)”**

Suppose we have

\[\{\psi\}(\sigma', O')\]

and

\[\{\psi\}(\sigma', O') \leadsto (\sigma', O')\]

we need to show that \(\{\psi\}(\sigma, O)\) holds where \(\psi' = \exists o \bullet le \mapsto o \land (\bigwedge_{i=0}^k (\text{type}(o) = T_i) \land f_i(\psi))\) is the weakest precondition for the formula \(\psi\) defined in this paper.

From the definition of the operational semantics for method invocation, we can get the fact that \(le\) denotes some object and \(T'\) is just the object type of \(le\), and the method body of \(m\) invoked is \(e\) presented in operational semantics. So \(\exists o \bullet le \mapsto o \land (\bigwedge_{i=0}^k (\text{type}(o) = T_i) \land f_i(\psi))\) exists. By induction hypothesis, we can get that \(\{f'_i(f_i(\psi))\}(\sigma, O)\). And followingly, \(\exists o \bullet le \mapsto o \land (\bigwedge_{i=0}^k (\text{type}(o) = T_i) \land f_i(\psi))\) holds. Then the conclusion \(\{\psi\}(\sigma, O)\) holds.

**Case Object Creation. “N.new(x, \tau)”**

Suppose we have

\[\{N.new(x, \tau), (\sigma, O)\} \leadsto (\sigma'', O'')\]

and

\[\{\psi\}(\sigma'', O'')\]

What we need to show is that \(\{\psi\}(\sigma, O)\), where \(\psi\) is the weakest precondition defined in this paper.

We can consider the execution of object construction as two steps: The first one create a new object \(o_0(N)\) and assign it to \(x\), the second one initiate \(o_0(N)\) by executing the commands in the constructor. By induction hypothesis, we can get the fact \(\{f'(f(\psi))\}(\sigma', O')\), in which \(\sigma' \equiv [\sigma|x : r]\), and \(O' \equiv [O]r \mapsto (\text{attr}(N) : \text{rnul})\). So we can get for every \(x'\), \(\{f'(f(\psi))[x'|x]\}(\sigma^0, O'^0)\), where \(\sigma^0 \equiv [\sigma|x' : r]\), and \(O'^0 \equiv [O]r \mapsto (\text{attr}(N) : \text{rnul})\). Because of the definition of \(+\), we have that \(\{\psi\}(\sigma, O)\).
– **Case Object Destruction, “**free(le)**”**

Similar to the proof of the soundness, we only prove the first rule of weakest precondition semantics for object destruction is complete, while the idea of proving the other one is similar. Suppose we have

\[
(free(le), (\sigma, O)) \leadsto (\sigma'', O'')
\]

and

\[
\{ \psi \}(\sigma'', O'')
\]

What we need to show is that \( \{ \psi' \}(\sigma, O) \), where \( \psi' \) is the weakest precondition defined in this paper.

By induction hypothesis, we can get the fact that \( \{ f(\psi) \}(\sigma', O') \), where \( \sigma', O' \) are as defined in the operational semantics for object destruction. By the definition of the separation operator \( * \), the conclusion \( \{ (le \mapsto -) \ast f(\psi) \}(\sigma, O) \) holds.

The proofs for structural commands are similar to the ones in ordinary imperative languages. Here we list one of them formally.

– **Case Sequential Composition, “**c_1;c_2**”**

Assume \( \{ \psi \}(\sigma'', O'') \) and \( (c_1;c_2, (\sigma, O)) \leadsto (\sigma'', O'') \). What we have need to prove is that \( \{ \psi' \}(\sigma, O) \), where \( \psi' \) is the weakest precondition defined in this paper. By the induction hypothesis, we can get the conclusion very easily.

### 8 Refinement

One of the most significant usages of a WP semantics is that one can define the notion of Refinement based on it. In this section, we propose our definitions of refinement and give some examples.

#### 8.1 Definitions

The definition of **Command Refinement** states that a command \( c_1 \) can be replaced by another command \( c_2 \) in the same environment.

**Definition 3 (Command Refinement).** For a sequence of given class declarations \( cds \), we denote its corresponding typing environment as \( \Gamma_{cds} \). For two pieces of code \( c_1 \) and \( c_2 \) in the same class \( C \), we give the definition of refinement as follows:

\[
\Gamma_{cds}, C, m \triangleright c_1 \subseteq c_2 \overset{\text{def}}{=} \forall \psi \bullet [\Gamma_{cds}, C, m \triangleright c_1 : \text{com}]\psi \Rightarrow [\Gamma_{cds}, C, m \triangleright c_2 : \text{com}]\psi
\]

**Data Refinement** is an extended notion of command refinement. Following is the definition of data refinement of commands in a given class:
Definition 4 (Data Refinement). For a given typing environment $\Gamma$ and a class $C$, suppose $\alpha_1, \alpha_2$ are the free variables of $c_1, c_2$ respectively, and $I$ is the linking invariant between $\alpha_1$ and $\alpha_2$, we give the definition of data refinement inside the class $C$ as follows:

$$\Gamma, C, m \triangleright c_1 \sqsubseteq_I c_2 \overset{\text{def}}{=} (\exists \alpha_1 \bullet I \land [\Gamma, C, m \triangleright c_1](\psi))$$

$$\Rightarrow [\Gamma, C, m \triangleright c_2]\left(\exists \alpha_1 \bullet I \land \psi\right)$$

The most important notion is Class Refinement which stands for that a sequence of classes offers no fewer services to the client code than another. A sequence of class declarations refines another if it can do anything the other can.

Definition 5 (Class Refinement). For two sequences of class declarations $cds_1$ and $cds_2$ which do not contain `main()` method, we give the definition of refinement as follows:

$$cds_1 \sqsubseteq cds_2 \overset{\text{def}}{=} \forall \psi, le, C_1 \in cds_1, m_1, m, \bar{r}_1 \bullet$$

$$\exists C_2 \in cds_2, m_2, \bar{m}, \bar{r}_2 \bullet$$

$$([\Gamma_{cds_1}, C_1, m_1 \triangleright le.m(\bar{r}_1) : \text{com}]\psi)$$

$$\Rightarrow [\Gamma_{cds_2}, C_2, m_2 \triangleright le.\bar{m}(\bar{r}_2) : \text{com}]\psi$$

Sometime when the visibility of $m$ is the whole class, we ignore $m_1$ and $m_2$.

Please notice the differences between our definitions and the corresponding ones in [7] or [16]. Because we do not bind the classes to the main method, a wider scope of the refinement is permitted. This makes it easier to cover many practical program transformations. We will illustrate this in one of the later examples.

8.2 Examples

We give two examples of refinement as follows:

**Parameterize Method.** This example comes from M. Fowler’s book on refactoring [13]. It says that when several methods do similar things but with different values contained in the method body, we can create one method that uses a parameter for the different
values.

```latex
cds_1 : cds;
    class M{
        m_A()\{c[true/x]\};
        m_B()\{c[false/x]\}
    }

cds_2 : cds;
    class M{
        m(\text{bool} \ \text{stamp})\{c[stamp/x]\}
    }
```

where \(c\) is composed of a sequence of commands which do not contain invocation to \(m_A()\) or \(m_B()\) and the other methods in \(cds\) do not invoke \(m_A()\) or \(m_B()\).

In our model, we have \(cds_1 \subseteq cds_2\) without any other condition required for main program.

The content is illustrated in Figure 1. To prove the refinement relation, we check the condition defined by Definition 5, as follows.

According to the definition, for a given well formed predicate \(\psi\) and left expression \(le\), we have to check for all \(C_1, m\), there are corresponding \(C_2, \hat{m}\) to satisfy the implication formula. However, the programs on each side are the same except for \(m_A()\) and \(m_B()\) in class \(M\). So the only cases making sense are as follows:

1. For \(m_A()\), we should have
   \[
   \exists C_2 \in cds_2, \hat{m}, \tau_2 \cdot
   ([\Gamma_{cds_1}, C_1 \triangleright le.m_A() : \text{com}] \psi
   \Rightarrow [\Gamma_{cds_2}, C_2 \triangleright le.\hat{m}(\tau_2) : \text{com}] \psi
   \]
   and
2. For \(m_B()\), we should have
   \[
   \exists C_2 \in cds_2, \hat{m}, \tau_2 \cdot
   ([\Gamma_{cds_1}, C_1 \triangleright le.m_B() : \text{com}] \psi
   \Rightarrow [\Gamma_{cds_2}, C_2 \triangleright le.\hat{m}(\tau_2) : \text{com}] \psi
   \]

Because they are quite similar, we only give the detailed proof for the first case. Take \(C_1\) itself as \(C_2\), \(m\) as \(\hat{m}\), and \(true\) as \(\tau_2\).

Suppose we have
\[
\psi' = [\Gamma_{cds_1}, C_1 \triangleright m_A() : \text{com}] \psi
\]

In our semantic model, the only law that can infer the above result is METHOD INVOCATION. Thus, the hypothesis of the law must hold. Because \(c[true/x]\) is the method body of \(m_A()\), we have the following results:
\[
type(O(\gamma_{<\sigma,O>}(le))) = M
\]

and
\[
\psi' = [\Gamma_{cds_1}, M \triangleright (c[true/x])[le/self] : \text{com}] \psi
\]

The two substitutions are independent, thus we can combine them:
\[
\psi' = [\Gamma_{cds_1}, M \triangleright c[true/x, le/self] : \text{com}] \psi
\]
On the other hand, using the METHOD INVOCATION law, we have
\[
\Gamma_{cds_2}, C_2 \triangleright le.\hat{m}(\tau_2) : \text{com}]\psi
= \Gamma_{cds_2}, M \triangleright e[\text{true}/x, le/self] : \text{com}][\psi
\]

There is no typing problem if we substitute \(cds_1\) to \(cds_2\) because \(e\) does not contain any invocation to \(m_A()\) or \(m_B()\). Thus \(\Gamma_{cds_2}, M \triangleright e : \text{com}\) is equivalent to \(\Gamma_{cds_1}, M \triangleright e : \text{com}\). So we have
\[
\Gamma_{cds_2}, M \triangleright e[\text{true}/x, le/self] : \text{com}][\psi
= \Gamma_{cds_1}, M \triangleright e[\text{true}/x, le/self] : \text{com}][\psi
= \psi'
\]

Thus we reach the end of the proof. \(\square\)

Delegation Across Classes. In this example we present a refactoring law in [13] which has strong background in incremental software development. The law says that, a class can delegate some of its tasks to another new class. Here are the details:

\(cds_1: cds;\)
\[\text{class } M \{ \text{prot } x : T; m()\{e\}\}\]

\(cds_2: cds;\)
\[\text{class } N \{ \text{prot } x : T; m()\{e\}\};\]
\[\text{class } M \{ \text{prot } n : N; m()\{m()\}\}\]

Now we prove \(cds_1 \subseteq cds_2\).

Here only the method body of \(m\) in class \(M\) is modified. So to verify \(cds_1 \subseteq cds_2\), what we need to check is, for all \(\psi\) and \(le\),
\[\exists C_2 \in cds_2, \hat{m}, \tau_2 \in (\Gamma_{cds_1}, C_1 \triangleright le.m() : \text{com}]\psi
\Rightarrow [\Gamma_{cds_2}, C_2 \triangleright le.\hat{m}(\tau_2) : \text{com}][\psi
\]

We just take \(C_1, m\) as \(C_2, \hat{m}\). (Here \(m()\) has no parameter, so \(\tau_2\) does not exist.) Thus what we need to prove is:
\[\Gamma_{cds_1}, C_1 \triangleright le.m() : \text{com}]\psi
\Rightarrow [\Gamma_{cds_2}, C_1 \triangleright le.m() : \text{com}][\psi

34
By expending the formulae via the METHOD INVOCATION law, we have

\[
\begin{align*}
\Gamma_{cds_1}, C_1 \triangleright le.m() : \text{com}\psi \\
\Gamma_{cds_1}, M \triangleright e[le/self] : \text{com}\psi
\end{align*}
\]

and

\[
\begin{align*}
\Gamma_{cds_2}, C_1 \triangleright le.m() : \text{com}\psi \\
\Gamma_{cds_2}, N \triangleright e[le/self] : \text{com}\psi
\end{align*}
\]

The class \(M\) in \(cds_1\) has the same content with the class \(N\) in \(cds_2\). So the well-formedness \(e\) is the same in both classes, i.e., \(\Gamma_{cds_1}, M \triangleright e[le/self] : \text{com}\) is equivalent to \(\Gamma_{cds_2}, N \triangleright e[le/self] : \text{com}\).

Thus,

\[
\begin{align*}
\Gamma_{cds_1}, M \triangleright e[le/self] : \text{com}\psi \\
\Gamma_{cds_2}, N \triangleright e[le/self] : \text{com}\psi
\end{align*}
\]

So we can conclude \(cds_1 \subseteq cds_2\)

\[\Box\]

9 Conclusions and Future Work

In this paper, by defining an extended storage model and the corresponding assertions semantics, we presented an extended and revised Separation Logic, the OO Separation Logic with which the local reasoning for OO programs could be carried on. Also we presented a typing environment which is enlightened by, and similar to, but more powerful and precise than the one proposed in [7]. Supported by the new logic and typing environment, motivated by advancing WP semantic theories defined by former researchers [11, 22, 7], we have proposed a weakest precondition semantics for object-oriented languages with reference types. Given a clear comparison to existing work [7] and [10], we conclude that our WP semantics captures the essentials of object-orientation better. Further, in our WP semantic model, we defined program transformation in terms of Refinement [22]. With the examples presented, we showed that, supported by our definitions, it is easier to model many practical program transformations in a reasonable way.

As for the future research directions, we would like to do the following:

The immediate future work is to prove the soundness and completeness properties of our semantics. Here, for a particular rule, it is sound means that a state which satisfies the precondition \(\vdash \psi\) can really reach a state satisfying the assertion \(\psi\) after the execution. While a rule is complete means that for any other \(\psi'\), if a state satisfying \(\psi'\) can reach \(\psi\) after the execution, it must also satisfy \(\vdash \psi\). In [18], S. Ishtiaq and P.W. O’Hearn proved the soundness and completeness properties of their backwards reasoning rules in traditional Separation Logic (in their THEOREM 4). We believe that the corresponding results hold for the OO Separation Logic. Similar to [18], we will give the operational rules for each command in terms of the modifications of \(\sigma\) and \(O\), and then prove our soundness and completeness w.r.t the operational semantics.

Having given the notions of refinement which is wider than the existing work, we realize that there are still many useful techniques in software development which can not be formalized by our definition in a clear way, i.e., a form without too many side
conditions. It seems that the assertion language is not abstract enough which makes us have to manage many details of programs. We plan to add the Abstract Predicate in [27] to our assertions to capture the abstraction concerns in software development.

Another natural future work is to extend our language to support full Java. We will try to extend our model for more complicated object oriented features such as static method, exception handling and multi-threads.

Knowing the read-write behavior of code will enable a number of useful optimizations of programs. Effect Systems [15, 8] are invented for this purpose. Recently, G.M. Bierman, M.J. Parkinson, and A.M. Pitts investigated an effect system, MJ e, for a subset of Java in [3]. We would like to develop an effect system for our language in conjunction of OO Separation Logic. Due to the local reasoning ability of Separation Logic, hopefully, the effect system could localize the read-write effects, thus speed up the effect inference.

With the support of this formalism, we will work to formalize the refactorings in [13] and pattern directed refactorings in [19] based on the semantics with reference types. This is what [9] would like, but did not totally succeed to do.

We realize that, in many cases, the proofs in the model of weakest preconditions are hard to be achieved by hand. The introducing of Separation Logic assertions complicates the formula of predicates, thus makes this problem heavier. We are thinking about developing an interactive theorem prover, or an approach for using existing provers, to support the process of generating the needed proofs.

Acknowledgement. Quan Long would like to show his great appreciations to Jifeng He and Zhiming Liu who introduced him the articles [7, 5] and their earlier work [16].

References

14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.