A Separation Logic for OO programs*

Yijing Liu and Qiu Zongyan

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

Abstract. We present a general storage model that reflects features of object oriented (OO) languages with pure reference semantics. Based on this model, we develop an OO Separation Logic (OOSL) to specify and verify OO programs. Many inference rules in the Separation Logic still hold in OOSL. Additionally, OOSL has certain properties important to OO reasoning. We introduce Hoare-Triple for a small OO language, and use the Schorr-Waite Marking Algorithm as a verification example.

Keywords: Object Orientation, Separation Logic, Verification

1 Introduction

Object-orientation (OO) paradigm is and will remain important for software development and programming languages design, because it supports many very useful abstractions. However, many new challenges in program specification and verification present in the OO field. There are two key issues mutually depending on each other: (1) building proper formal models for OO languages, and (2) developing useful methods to specify and verify OO programs. Researchers have proposed many formal frameworks to describe core concepts of OO programs.

As the basis for formal studies, various state models are proposed to represent complicated structures of state space of OO programs. Major models can be roughly classified as *Object Graph Model*, *Access Trace Model*, and *Stack Heap Model*.

The Object Graph Models treat objects state as some form of graphs. Examples in this direction include the topological model [12], or object diagram [17]. In the graph, vertexes denote objects, and edges denote variables and object attributes (i.e., instance variables). Models of this kind are intuitive and always independent of languages. [6] presents an operational semantics based on a graph model. However, a suitable reasoning framework for graph models still does not exist. The Access Trace Model (originally for pointer-programs) was introduced by [4], where each object was identified by a set of traces to the object. Access Trace models have advantages in alias analysis [2], but seem too abstract for general purpose. [3] attempted to define a general inference framework for a trace model. Stack Heap Models are extensions of normal store model, with an additional heap (a map from address to values) to represent objects. Stack Heap

* Supported by NNSFC Grant No. 60718002 and No. 60773161.

Models seem low-level, however, they are relatively easy to used for definitions of program semantics. Some works have been done upon such models, e.g. [13]. However, a full accounting of all important OO features is still missing.

On the other hand, many works have been done on the specification and verification of OO programs. Although JML [7] and Spec# [1] catch increasing attentions, many critical issues of OO programs, especially that related to the mutable object structures, have not been considered deeply there. Many semantic issues must be investigated and understood for a big-leap in this field. [8] gives a comprehensive overview for the achievements and challenges in this area.

Separation Logic [15] is a powerful tool to handle shared mutable data structures. Many verification techniques based on it have been developed, mainly for C-like programs, and some targeting OO programs. However, it is not straightforward to use the Separation Logic to specify and verify OO programs, because the underlying storage model of the logic is not ready for many OO concepts. Especially, there is no correspondents of object attributes in the model. Some researchers tried to revise the Separation Logic targeting OO fields [9, 14]. The work presented here is also in this direction. We will discuss and compare these works in Section 6.

In this article, we proposed a model for the object pool (heap), with a novel definition for the separation of object pools. The model provides a clear concept for objects. Empty objects can be naturally represented and reasoned. We develop a revised Separation Logic, named OO Separation Logic (OOSL), for expressing OO program states. User-defined predicates and logic environment are clearly defined, and the semantics of the logic is defined as a least fix-point which is guaranteed existing. The logic adopts classical semantics, thus is more expressive than the logic with intuitionistic semantics (referring to [5]) as used in [14]. Properties of OOSL are explored, especially a new concept named *separated assertions*, which is useful in reasoning OO programs. Due to the classical nature, properties of OO programs can be precisely specified and verified. We introduce a simple OO language, and develop Hoare-Triple like inference rules based on OOSL. We use Schorr-Waite Marking Algorithm as an example to show how to specify and verify OO programs with OOSL.

The rest of the paper is organized as follows: We introduce an OO storage model in Section 2. OOSL is developed in Section 3. We introduce a simple OO language and its inference rules in Section 4. In Section 5, we study the Schorr-Waite Marking Algorithm. Finally, we discussed some related works and future research directions.

2 An OO Storage Model

Now we introduce a storage model for OO programs with pure reference nature.

The model is defined based on three basic sets Name, Type and Ref, these sets model basic concepts, such as variables, fields, types and references, in OO program.

- Name: an infinite set of names, used for naming various entities, e.g., constants, variables, attributes, etc. Three special names, $\mathbf{true}, \mathbf{false}, \mathbf{null} \in \mathrm{Name}$, denote boolean and null constants.
- Type: an infinite set of types, including predefined types and user-defined types (or called classes). Subtype relation is represented by symbol $<:$, where $T_1 <:$

T_2 states that T_1 is a subtype of T_2. We assume there are three predefined types $\mathbf{Object}, \mathbf{Null}$ and $\mathbf{Bool}$. $\mathbf{Object}$ is the super type of all classes. $\mathbf{Null}$ is the subtype of all classes. And $\mathbf{Bool}$ is the type of boolean objects. Given a type T, we can obtain its attributes by function $\mathsf{attrs} : \mathrm{Type} \to \mathrm{Name} \to \mathrm{Type}$; and we define $\mathsf{attrs}(\mathbf{Object}) = \mathsf{attrs}(\mathbf{Null}) = \mathsf{attrs}(\mathbf{Bool}) = \emptyset$. Other predefined types, such as Integer, can be added easily, but we consider only boolean type here.

– Ref: an infinite set of references which are the identities of objects. Corresponding to the Name constants, Ref contains three basic references rtrue, rfalse and rnull, where rtrue, rfalse refer two $\mathbf{Bool}$ objects, and rnull never refers to any object. We assume two primitive functions on Ref:[1]
  - $\mathsf{eqref} : \mathrm{Ref} \to \mathrm{Ref} \to \mathsf{bool}$, justifies whether two references are same, i.e. given references $r_1, r_2 \in \mathrm{Ref}$, $\mathsf{eqref}(r_1, r_2) = \mathsf{true}$ iff r_1 is same to r_2.
  - $\mathsf{type} : \mathrm{Ref} \to \mathrm{Type}$, decides the runtime type of object referred by reference. We define $\mathsf{type}(\mathsf{rtrue}) = \mathsf{type}(\mathsf{rfalse}) = \mathbf{Bool}, \mathsf{type}(\mathsf{rnull}) = \mathbf{Null}$.

In fact, Name, Type and functions (relations) defined on them, such as dtype, $<:$ and attrs, make up the static information of an OO program.

Based on above concepts, we define an OO storage model. It is similar to the classical Stack-Heap model with two components:

$$\mathrm{Store} \ \widehat{=}\ \mathrm{Name} \rightharpoonup_{\mathsf{fin}} \mathrm{Ref} \qquad \mathrm{Opool} \ \widehat{=}\ \mathrm{Ref} \rightharpoonup_{\mathsf{fin}} \mathrm{Name} \rightharpoonup_{\mathsf{fin}} \mathrm{Ref}$$
$$\mathrm{State} \ \widehat{=}\ \mathrm{Store} \times \mathrm{Opool}$$

where notation "$\rightharpoonup_{\mathsf{fin}}$" denotes finite partial functions.

We will use σ and O, possibly with subscript, to denote elements of Store and Opool respectively. A store $\sigma \in \mathrm{Store}$ maps variables and constants to references, and an object pool $O \in \mathrm{Opool}$ maps references to field-reference pairs. A runtime state s is a pair, $s = (\sigma, O) \in \mathrm{State}$, consisting of a store and an object pool. For every $\sigma \in \mathrm{Store}$, we assume that $\sigma\mathbf{true} = \mathsf{rtrue}$, $\sigma\mathbf{false} = \mathsf{rfalse}$ and $\sigma\mathbf{null} = \mathsf{rnull}$.

We will use $r, r_1, \ldots$ to denote references, and $a, a_1, \ldots$ to denote attributes of objects. An element of O is a pair (r, f), where r is a reference to some object o, f is a function from attributes of o to their corresponding values (also references). When we mention the domain of O, we sometimes want to mean a subset of Ref associated with a set of objects as discussed above, or sometimes a subset of $\mathrm{Ref} \times \mathrm{Name}$ associated with a set of values (references). We use $\mathsf{dom}\, O$ for the first case. For the second case, we define notation $\mathsf{dom}_2\, O \ \widehat{=}\ \{(r, a) \mid r \in \mathsf{dom}\, O, a \in \mathsf{dom}\, O(r)\}$, that is, $\mathsf{dom}_2\, O$ gets all the reference and attribute pairs of non-empty objects in O.

When considering the program states, we need to ask for some regularity, that is, the well-typedness. Now we define the concept for states that are consistent with the static information of the program, i.e., the well-typed states. We assume a function $\mathsf{dtype} : \mathrm{Store} \to \mathrm{Name} \to \mathrm{Type}$, where $\mathsf{dtype}(\sigma)(v)$ gives the declaration type of constant or variable v in store σ.

Definition 1 (Well-typed Store). *A store σ is well-typed iff*

$$\forall v \in \mathsf{dom}\, \sigma \cdot \mathsf{type}(\sigma(v)) <: \mathsf{dtype}(\sigma)(v).$$

[1] For example, we can define every reference as a pair (t, id) where $t \in \mathrm{Type}$ and $id \in \mathbf{N}$, define eqref as pair equivalence, and $\mathsf{type}(r) = r.first$.

Clearly, this condition requires that all variables hold valid values.

Definition 2 (Well-typed Opool). *An Opool O is well-typed iff*

- $\forall (r,a) \in \mathsf{dom}_2\, O \cdot a \in \mathsf{Att}(r) \wedge \mathsf{type}(O(r)(a)) <: \mathsf{attrs}(r)(a)$, and
- $\forall r \in \mathsf{dom}\, O \cdot Att(r) = \emptyset \vee (\mathsf{Att}(r) \cap \mathsf{dom}\, O(r) \neq \emptyset)$.

where $\mathsf{Att}(r) \mathrel{\widehat{=}} \mathsf{dom}\, \mathsf{attrs}(\mathsf{type}(r))$.

Note that $\mathsf{attrs}(\mathsf{type}(r))$ is a function from an attribute set to Type. The first condition requires that all attributes are valid according to types of all objects in O, and all attributes hold values of correct types. The second condition requires that if a non-empty object (according to its type) is in O, then O must contains at least one attribute of the object. Thus we can identify empty objects in any Opool.

As an example, suppose we have statically $\mathsf{dom}\, \mathsf{attrs}(C) = \{a_1, a_2, a_3\}$, and have a program state where $\mathsf{type}(r_1) = \mathbf{Object}, \mathsf{type}(r_2) = C$. In this case, we easily know that $O_1 = \{r_1 \mapsto \emptyset, r_2 \mapsto \{a_1 \mapsto \mathsf{rnull}, a_2 \mapsto \mathsf{rnull}\}\}$ is a well-typed Opool, but $O_2 = \{r_1 \mapsto \emptyset, r_2 \mapsto \emptyset\}$ is not, because $\mathsf{type}(r_2) = C$ has attributes. Further, we can calculate that $\mathsf{dom}\, O_1 = \{r_1, r_2\}$, and $\mathsf{dom}_2\, O_1 = \{(r_2, a_1), (r_2, a_2)\}$.

Definition 3 (Well-typed State). *A state $s = (\sigma, O)$ is well-typed iff both σ and O are well-typed.*

We will only consider well-typed states from now on. This requirement makes sense because a well-typed program always runs under well-typed states, and the well-typedness can be checked statically based on the type system of the language.

For convenience, we will use notation $(r, \{\overline{(a,-)}\})$ to denote an (or a part of an) object, and use $(r, a, -)$ to denote a cell (state of an attribute of an object) in the Opool. Here "$-$" represents some value which we do not care about.

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

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

where the right $\oplus$ is the standard function overriding operator. Thus, for Opool O_1, $O_1 \oplus \{(r, a, r')\}$ gives a new Opool, where only one attribute value (the value for a) of the object pointed by r is modified (denoted by r').

We borrow some concepts and notations from the Separation Logic. $O_1 \perp O_2$ indicates that two Opools O_1 and O_2 are separated from each other. The formal definition for $\perp$ is new for separating object pools,

$$\begin{aligned} O_1 \perp O_2 \mathrel{\widehat{=}} {} & \forall r \in \mathsf{dom}\, O_1 \cap \mathsf{dom}\, O_2 \cdot \\ & O_1(r) \neq \emptyset \wedge O_2(r) \neq \emptyset \wedge \mathsf{dom}\,(O_1(r)) \cap \mathsf{dom}\,(O_2(r)) = \emptyset. \end{aligned}$$

That is, if a reference, referring to some object o, is in both $\mathsf{dom}\, O_1$ and $\mathsf{dom}\, O_2$, then both O_1 and O_2 must contain non-empty subsets of o's attributes, respectively (the well-typedness also guarantees this); and these two subsets must be disjoint. This means that we can separate attributes of an object in the Opool (providing that the object is not empty). Additionally, an empty object cannot be in two separated Opools at the same time, because it cannot be partitioned. We will use $O_1 * O_2$ to indicate the union of O_1 and O_2, $O_1 \oplus O_2$, when $O_1 \perp O_2$.

As an example, suppose,

$$\begin{aligned} &O_1 = \{(r_1, \emptyset), (r_2, \{(a_1, \mathsf{rnull})\})\},\ \ O_2 \ = \ \{(r_2, \{(a_2, \mathsf{rnull})\})\}, \\ &O_3 = \{(r_1, \emptyset), (r_2, \{(a_2, \mathsf{rnull})\})\}. \end{aligned}$$

We have $O_1 \perp O_2$, although each of them contains a part of object pointed by r_2. But $O_1 \not\perp O_3$ because $r_1 \in \mathsf{dom}\, O_1 \cap \mathsf{dom}\, O_3$, while $O_1(r_1) = \{\}$. Additionally, $O_2 \not\perp O_3$, because $r_2 \in \mathsf{dom}\, O_2 \cap \mathsf{dom}\, O_3$, while $\mathsf{dom}\,(O_2(r_2)) \cap \mathsf{dom}\,(O_3(r_2)) = \{a_2\}$.

Clearly, above definition of separation takes the basic cell (r, a, r') as a unit, but it also offers a careful treatment for empty objects. It is a revision of the separation concept in Separation Logic, while also takes into account the characteristics of OO programs. This definition plays an important role in our work.

3 An OO Separation Logic

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

3.1 Assertions Language

We use Ψ for the set of all assertions of OOSL, and $\psi, \psi_1, \psi_2 ...$ as typical assertions. The assertion language of OOSL is similar to what in Separation Logic, with some revisions and extensions, to fit the special needs of OO programs.

Basic assertions are of two kinds in OOSL, namely *primitive assertions* and *user-defined assertions*. All assertions are built on them.

Primitive assertions have the forms defined by the following rules:

$$\begin{aligned} \alpha &::= \mathbf{true} \mid \mathbf{false} \mid v = r \mid r_1 = r_2 \\ \beta &::= \mathbf{emp} \mid r_1.a \mapsto r_2 \mid \mathsf{obj}(r, T) \end{aligned}$$

where v is a variable or constant name, r denotes references. In fact, here r servers as both "references" and "reference variables" (logic variables) at the same time.

As shown, primitive assertions fall into two categories, where

– α denotes a kind of assertions that are independent of Opools. References are atomic values in our logic. For any two references r_1, r_2, $r_1 = r_2$ holds iff r_1 and r_2 are identical, i.e., $\mathsf{eqref}(r_1, r_2)$. We treat $r = v$ the same as $v = r$.

[2] One exception might be variables and attributes of primitive types, while many languages use value model for them for efficiency.

– β denotes assertions involving Opools. Empty and singleton assertions take the similar forms as in Separation Logic. As we said before, a cell in Opool is an attribute-value binding of an object (denoted by a reference), thus the singleton assertion takes the form $r_1.a \mapsto r_2$. To make OOSL clear and simple, we do not define $v.a \mapsto \ldots$ as a primitive assertion, because it is not really primitive. Certainly, we can define $v.a \mapsto r$ as $\exists r' \cdot v = r' \wedge r.a \mapsto r'$.
– We add an assertion form $\mathsf{obj}(r, T)$ to indicate that r refers to a complete object of type T, and the Opool only contains this object. In Separation Logic, people use $l \mapsto -$ or $l \hookrightarrow -$ to denote that location l is allocated in current heap. Because the existence of empty object, we cannot use $r.a \mapsto -$ or $r.a \hookrightarrow -$ to express that object which r refers to is allocated in current Opool. To solve this problem, we introduce assertion form $\mathsf{obj}(r, T)$ in OOSL. We will use $\mathsf{obj}(r, -)$ when we do not care about r's type.

We allow users to define new predicates in OOSL. In fact, people always need to define some recursive predicates to support specification and verification of OO programs involving recursive data structures, e.g., list, tree, etc.

These definitions are recorded in a *Logic Environment* Λ with the form defined by:

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

where ε denotes the empty environment, p is a symbol (predicate name) selected from a given set $\mathcal{S}$, $\overline{r}$ are (a list of) formal parameters, and ψ is the body, which is an assertion correlated with $\overline{r}$. Recursive definitions are allowed.

As a well-formed logic environment, we ask for that Λ must be self-contained, that is: The body ψ of a definition in Λ cannot use symbols not defined in Λ. Further, we require that Λ must be *finite* and *syntactically monotone*[3], then a fix-point semantics for Λ exists.

For every symbol p defined in Λ, we use $\mathsf{argc}_\Lambda(p)$ to denote its arguments number, where subscript Λ may be omitted when there is no ambiguity.

Complex assertions are built upon basic assertions with classical FOL combinators and separation combinators from Separation Logic:

$$\psi ::= \alpha \mid \beta \mid p(\overline{r}) \mid \neg\psi \mid \psi \vee \psi \mid \psi * \psi \mid \psi -\!\!* \psi \mid \exists r \cdot \psi$$

where $p(\overline{r})$ is a user-defined assertion with real arguments $\overline{r}$.

Please notice that only references, but not variables, can be quantified. The intension is clear: variables are defined in the program text, thus are free variables in assertions.

We will use $\psi[v/x]$ (or $\psi[r/x]$) to denote the assertion built from ψ by substituting variable x with variable or constant v (reference r) in it. And $\psi[r_1/r_2]$ denotes the assertion build from ψ by substituting r_2 with r_1.

At last, we define some abbreviations, that are classical:

$$\begin{array}{ll}
\psi_1 \wedge \psi_2 \equiv \neg(\neg\psi_1 \vee \neg\psi_2) & \psi_1 \Rightarrow \psi_2 \equiv \neg\psi_1 \vee \psi_2 \\
\forall r \cdot \psi \quad\ \equiv \neg\exists r \cdot \neg\psi & \\
r.a \mapsto - \ \equiv \exists r' \cdot r.a \mapsto r' & r.a \hookrightarrow r' \equiv r.a \mapsto r' * \mathbf{true}
\end{array}$$

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

$$
\begin{array}{lll}
\mathcal{M}_{\mathcal{I}}(\mathbf{false}) & = \emptyset & \text{(I-FALSE)} \\
\mathcal{M}_{\mathcal{I}}(\mathbf{true}) & = \mathrm{State} & \text{(I-TRUE)} \\
\mathcal{M}_{\mathcal{I}}(v = r) & = \{(\sigma, O) \mid \sigma(v) = r\} & \text{(I-LOOKUP)} \\
\mathcal{M}_{\mathcal{I}}(r_1 = r_2) & = \mathrm{State} \quad \text{iff } \mathsf{eqref}(r_1, r_2) & \text{(I-REF-EQ)} \\
\mathcal{M}_{\mathcal{I}}(r_1 = r_2) & = \emptyset \quad \text{iff } \neg\mathsf{eqref}(r_1, r_2) & \text{(I-REF-NEQ)} \\
\mathcal{M}_{\mathcal{I}}(\mathbf{emp}) & = \{(\sigma, \emptyset)\} & \text{(I-EMPTY)} \\
\mathcal{M}_{\mathcal{I}}(r_1.a \mapsto r_2) & = \{(\sigma, \{(r_1, a, r_2)\})\} & \text{(I-SINGLE)} \\
\mathcal{M}_{\mathcal{I}}(\mathsf{obj}(r, T)) & = \{(\sigma, O) \mid \mathsf{type}(r) = T \wedge \mathsf{dom}\, O = \{r\} \wedge & \text{(I-OBJ)} \\
 & \qquad \mathsf{dom}\,(O(r)) = \mathsf{dom}\,(\mathsf{attrs}(T))\} & \\
\mathcal{M}_{\mathcal{I}}(p(\overline{r})) & = \mathcal{I}(p)(\overline{r}) & \text{(I-APP)} \\
\mathcal{M}_{\mathcal{I}}(\neg\psi) & = \mathrm{State} \setminus \mathcal{M}_{\mathcal{I}}(\psi) & \text{(I-NEG)} \\
\mathcal{M}_{\mathcal{I}}(\psi_1 \vee \psi_2) & = \mathcal{M}_{\mathcal{I}}(\psi_1) \cup \mathcal{M}_{\mathcal{I}}(\psi_2) & \text{(I-OR)} \\
\mathcal{M}_{\mathcal{I}}(\psi_1 * \psi_2) & = \{(\sigma, O) \mid \exists O_1, O_2 \cdot O_1 * O_2 = O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{I}}(\psi_1) & \text{(I-S-CONJ)} \\
 & \qquad \wedge (\sigma, O_2) \in \mathcal{M}_{\mathcal{I}}(\psi_2)\} & \\
\mathcal{M}_{\mathcal{I}}(\psi_1 \mathrel{-\!\!*} \psi_2) & = \{(\sigma, O) \mid \forall O_1 \cdot O_1 \bot O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{I}}(\psi_1) & \text{(I-S-IMPLY)} \\
 & \qquad \text{implies } (\sigma, O_1 * O) \in \mathcal{M}_{\mathcal{I}}(\psi_2)\} & \\
\mathcal{M}_{\mathcal{I}}(\exists r \cdot \psi) & = \{(\sigma, O) \mid \exists r \in \mathrm{Ref} \cdot (\sigma, O) \in \mathcal{M}_{\mathcal{I}}(\psi)\} & \text{(I-EX)}
\end{array}
$$

Fig. 1. Semantic function with interpretation $\mathcal{I}$

The last two abbreviations are widely used in Separation Logic related papers.

3.2 Semantics

Now, we provide a *Least Fix-point Semantics* for OOSL. We will define a semantic function which maps every assertion $\psi \in \Psi$ to a subset of State. To achieve this goal, we first define a formal semantics for Λ.

We introduce a family of *Predicate Functions*. For any $n \geq 0$, we define $\mathcal{P}_n \mathrel{\widehat{=}} \mathrm{Ref}^n \to \mathbb{P}(State)$, the set of functions from n references to subsets of State. Here n is the arity of the functions in $\mathcal{P}_n$. We define $\mathcal{P} \mathrel{\widehat{=}} \bigcup_n \mathcal{P}_n$, which is the set of all possible predicate functions. We introduce a function $\mathsf{arity} : \mathcal{P} \to \mathbf{N}$ to extract the arity of given predicate function: For any $p \in \mathcal{P}$, $\mathsf{arity}(p) = n$ iff $p \in \mathcal{P}_n$.

We will use p, q, possibly with subscripts, for the typical elements of $\mathcal{P}$. Given $p(\overline{r}), q(\overline{r'}) \in \mathcal{P}_n$, we define $p \leq q$ iff $\forall r_1, ..., r_n \cdot p(r_1, ..., r_n) \subseteq q(r_1, ..., r_n)$. Clearly, $(\mathbb{P}(State), \subseteq)$ forms a complete lattice, with $\emptyset$ and State as its bottom and top elements. So for any n, $(\mathcal{P}_n, \leq)$ is a complete lattice, with $\bot_{\mathcal{P}_n} = \{(r_1, ...r_n) \mapsto \emptyset\}, \top_{\mathcal{P}_n} = \{(r_1, ...r_n) \mapsto \mathrm{State}\}$ as its bottom and top elements.

With Predicate Functions, we define interpretations of Λ as follows.

Definition 4 (Interpretation of Logic Environment). *Given a logic environment Λ, we say a function $\mathcal{I} : \mathcal{S} \to \mathcal{P}$ is an interpretation of Λ iff for every symbol p defined in Λ, $p \in \mathsf{dom}\, \mathcal{I}$ and* $\mathsf{arity}(\mathcal{I}(p)) = \mathsf{argc}_{\Lambda}(p)$.

We use $\mathcal{I}_\Lambda$ to denote all interpretations of Λ. For any $\mathcal{I}_1, \mathcal{I}_2 \in \mathcal{I}_\Lambda$, we define:

$$\mathcal{I}_1 \leq \mathcal{I}_2 \text{ iff } \forall p \in \mathsf{dom}\,\Lambda \cdot \mathcal{I}_1(p) \leq \mathcal{I}_2(p).$$

Obviously, $(\mathcal{I}_\Lambda, \leq)$ is a complete lattice. $\bot_\Lambda = \{(p, \bot_{\mathcal{P}_{\mathsf{argc}_\Lambda(p)}}) | p \in \mathsf{dom}\,\Lambda\}$ is the bottom element, and $\top_\Lambda = \{(p, \top_{\mathcal{P}_{\mathsf{argc}_\Lambda(p)}}) | p \in \mathsf{dom}\,\Lambda\}$ is the top element.

We define a semantic function $\mathcal{M} : \mathcal{I} \to \Psi \to \mathbb{P}(\mathrm{State})$ for OOSL, the definition is presented in **Fig.1**. Note that $\mathcal{M}_\mathcal{I}$ means $\mathcal{M}(\mathcal{I})$ in the definition.

Clearly, a logic environment Λ can have many different interpretations, but not every interpretation makes sense. This leads the following definition.

Definition 5 (Model of Logic Environment). *Suppose $\mathcal{I}$ is an interpretation of Λ, we say $\mathcal{I}$ is a model of Λ iff for every definition $p(\overline{r}) \doteq \psi$ in Λ, we have:*

$$\forall \overline{r'} \cdot \mathcal{M}_\mathcal{I}(p(\overline{r'})) = \mathcal{M}_\mathcal{I}(\psi[\overline{r'}/\overline{r}]).$$

In fact, a model of Λ is a fix-point of function $\mathcal{N}_\Lambda : (\mathcal{S} \to \mathcal{P}) \to (\mathcal{S} \to \mathcal{P})$, which is defined as follows:

$$\mathcal{N}_\Lambda(\mathcal{I})(p) = \{(\overline{r'}, \mathcal{M}_\mathcal{I}(\psi[\overline{r'}/\overline{r}])\}, \quad \text{for any definition } p(\overline{r}) \doteq \psi \text{ in } \Lambda$$

The fix-point of $\mathcal{N}_\Lambda$ exists, because the self-containedness of Λ, and the syntactically monotonic requirement for each definition of symbols in Λ.

A given Λ may have many models. We choose the least one as its standard model, which is the *least fix-point* of $\mathcal{N}$. By Tarski's fix-point theorem, this standard model can be expressed as:

$$\mathcal{J}_\Lambda = \bigcup_{n=0}^{\infty} \mathcal{N}_\Lambda^n(\bot_\Lambda),$$

We give a simple example as an illustration. Suppose Λ contains only one definition

$$\mathit{list}(r) \doteq (r = \mathbf{null} \wedge \mathbf{emp}) \vee \exists r' \cdot (r.a \mapsto r') * \mathit{list}(r')$$

which describes lists linked on a. In order to get the standard model of Λ, we have:

$\mathcal{N}_\Lambda^0 = \bot_\Lambda$
$\mathcal{N}_\Lambda^1 = \{(\mathit{list}, \{(\mathbf{null}, \mathbf{emp})\})\}$
$\mathcal{N}_\Lambda^2 = \{(\mathit{list}, \{(\mathbf{null}, \mathbf{emp}), (r, r.a \mapsto \mathbf{null})\})\}$
$\mathcal{N}_\Lambda^3 = \{(\mathit{list}, \{(\mathbf{null}, \mathbf{emp}), (r, r.a \mapsto \mathbf{null})\}), (r, r.a \mapsto r' * r'.a \mapsto \mathbf{null})\})\}$
...

Then we get a model that describes all possible lists of this type.

With the standard model $\mathcal{J}_\Lambda$, we can define the formal semantics for our assertion language. We use $\sigma, O \models_\Lambda \psi$ to mean that ψ holds on state (σ, O) with respect to logic environment Λ. We have the following definition:

Definition 6 (Semantics of Assertions).

$$\sigma, O \models_\Lambda \psi \qquad \textit{iff} \qquad (\sigma, O) \in \mathcal{M}_{\mathcal{J}_\Lambda}(\psi).$$

We often use $\sigma, O \models \psi$ as a shorthand when Λ is not ambiguous.

3.3 Properties and Inference Rules

The semantics defined above have some good properties:

Lemma 1. *New predicate functions can be safely appended to Λ, without changing the meaning of existing symbols in Λ. Formally, if $\Lambda' = (\Lambda, p(\overline{r}) \doteq \psi)$, where p is not defined in Λ, we have for every symbol q defined in Λ:*

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

By this lemma, we can easily get:

Lemma 2. *Given a logic environment Λ:*
(1) we can safely append some new definitions to it, without changing semantics of symbols defined in Λ;
(2) if symbols $\overline{p}$ defined in Λ are not mentioned in other definitions in Λ, then we can safely remove them, without changing semantics of the other symbols defined in Λ. $\square$

And by OOSL's semantics, it is straightforward to prove the following propositions:

Lemma 3. *Suppose $\sigma, O \models \psi$, we have:*
(1) if $\mathsf{dom}\,\sigma' \cap \mathsf{dom}\,\sigma = \emptyset$, then $\sigma \cup \sigma', O \models \psi$;
(2) if ψ does not contain variables in σ', then $\sigma - \sigma', O \models \psi$. Here $\sigma - \sigma'$ denotes $\{(x, r) \in \sigma \mid x \notin \mathsf{dom}\,\sigma'\}$. $\square$

Lemma 4. $\sigma, O \models \psi[e/x]$, *if and only if* $\sigma \oplus \{x \mapsto \sigma e\}, O \models \psi$. $\square$

Lemma 5. *Suppose $a_1, a_2, ..., a_k$ are all attributes of type T, then we have:*

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

$\square$

Lemma 6. $\mathsf{obj}(r_1, -) * \mathsf{obj}(r_2, -) \Rightarrow r_1 \neq r_2$. $\square$

Lemma 7.

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

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

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

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

Many propositions in Separation Logic also hold in OOSL. For example, rules (axiom schemata) shown in the Section 3 of [15] are all valid here.

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

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

Definition 7 (Separated Assertions). *Two assertions ψ and ψ' are* separated *from each other, iff for all stores σ and Opools O, O', $\sigma, O \models \psi$ and $\sigma, O' \models \psi'$ implies $O \perp O'$.* $\square$

Lemma 8. *$r_1.a \mapsto -$ and $r_2.b \mapsto -$ are separated, provided that $r_1 \neq r_2$, or a and b are different attribute names.* $\square$

For example, suppose we have a $Node$ class with fields $value$ and $next$. For a reference $r : Node$, we know $r.value \mapsto -$ and $r.next \mapsto -$ are separated. No corresponding concept is in original Separation Logic, due to the absence of attributes.

Lemma 9. *Suppose ψ_1 and ψ_2 are separated.* (1) *If $\sigma, O_1 \models \psi_1$ and $\sigma, O_2 \models \psi_2$, then $\sigma, O_1 * O_2 \models \psi_1 * \psi_2$.* (2) *If $\sigma, O \models \psi_1 * \psi_2$, there exists an unique partition of $O = O_1 * O_2$, that $\sigma, O_1 \models \psi_1$ and $\sigma, O_2 \models \psi_2$.* $\square$

Lemma 10. *ψ_1 is separated from both ψ_2 and ψ_3, iff ψ_1 is separated from $\psi_2 * \psi_3$.* $\square$

Theorem 1. *For any ψ_1, ψ_2, ψ_3, if ψ_1 and ψ_2 are separated from each other, then $\psi_1 * (\psi_2 -\!\!* \psi_3) \Leftrightarrow \psi_2 -\!\!*(\psi_1 * \psi_3)$.*

Proof. The proof is as follows:

$\Rightarrow$: For any σ and O such that $\sigma, O \models \psi_1 * (\psi_2 -\!\!* \psi_3)$, there exist O_1, O_2, such that $O_1 * O_2 = O$, $\sigma, O_1 \models \psi_1$, and $\sigma, O_2 \models \psi_2 -\!\!* \psi_3$. By the definition of$-\!\!*$, for any O_3 satisfying $O_2 \perp O_3$,

$$\sigma, O_3 \models \psi_2 \text{ implies } \sigma, O_2 * O_3 \models \psi_3.$$

Because ψ_1 and ψ_2 are separated, then by **Lemma 9**,

$$\sigma, O_3 \models \psi_2 \text{ implies } \sigma, O_1 * O_2 * O_3 \models \psi_1 * \psi_3.$$

This is $\sigma, O \models \psi_2 -\!\!*(\psi_1 * \psi_3)$.

$\Leftarrow$: For any σ and O that $\sigma, O \models \psi_2 \mathrel{-\!\!*} (\psi_1 * \psi_3)$, for any O_1 that $O_1 \perp O$, if $\sigma, O_1 \models \psi_2$, then $\sigma, O_1 * O \models \psi_1 * \psi_3$. Now we fix this O_1. From $\sigma, O_1 * O \models \psi_1 * \psi_3$ we know there exist O_2 and O_3' such that $O_2 \perp O_3'$, $O_2 * O_3' = O_1 * O$, $\sigma, O_2 \models \psi_1$ and $\sigma, O_3' \models \psi_3$. Because ψ_1, ψ_2 are separated, then $O_2 \perp O_1$. Thus $O_3' = O_1 * O_3$ for some O_3. Now we have

$$\sigma, O_2 \models \psi_1,\ \sigma, O_1 \models \psi_2,\ \text{and}\ \sigma, O_1 * O_3 \models \psi_3.$$

Then we have $\sigma, O_3 \models \psi_2 \mathrel{-\!\!*} \psi_3$, because the choice of O_1 needs no extra restriction. Thus $\sigma, O \models \psi_1 * (\psi_2 \mathrel{-\!\!*} \psi_3)$, because $O = O_2 * O_3$. $\square$

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

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

3.4 Discussion

In this section, we discuss some expressiveness and extension issues about OOSL.

As presented above, we define a power assertion language for OOSL, especially the user-defined predicates, which notably enhance the expressiveness of OOSL. With OOSL, We can describe and infer recursive data structures, and some important properties between objects, such as accessibility, dangling and so on. Since our logic adopts classical semantics, it is more expressive than its intuitionistic cousin, e.g., what defined in [14]. We can use OOSL to describe the program state precisely, especially the Opool, i.e., what is in or is not in an Opool.

On the other hand, the primitive assertions of OOSL are very simple and specific, so we cannot describe quantitative relation or more complicated mathematical concepts with OOSL. But it is not difficult to extend OOSL to support these concepts. For example, if we want to support integer arithmetic in OOSL, we should

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

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

4 A Simple OO Language and Its Inference Rules

In this section, we study a simple OO language. For simplicity, we only consider basic commands here. High-level features, i.e., concepts related to method and class, involving more static structure and type information, will be studied in our further works. We demand that our storage model and OOSL are ready to deal with them.

The syntax of the language is as follows:

$$\begin{aligned}
e &::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid x \\
b &::= \mathbf{true} \mid \mathbf{false} \mid e = e \mid \neg b \mid b \wedge b \mid b \vee b \\
c &::= \mathbf{skip} \mid x := e \mid x.a := e \mid x_1 := x_2.a \mid \\
&\quad\; x := \mathbf{new}\ C() \mid c; c \mid \mathbf{if}\ b\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ c
\end{aligned}$$

where:

- x is a variable, C a class name or $\mathbf{Object}$, a an attribute name. We adopt restricted forms for expressions e, so their values depend only on the store. Complex expressions can be encoded with the help of auxiliary variables and assignments.
- Assignments are restricted to a number of special forms. Beside the plain assignment $x := e$, we have mutation assignment $x.a := e$, and lookup assignment $x_1 := x_2.a$. Other general cases can be also encoded by these forms. For instance, one can use $x := y.a$ and then refer to $x.a'$ as a replacement of $y.a.a'$.
- $x := \mathbf{new}\ C()$ creates a new raw object, that is this command do not initialize the new object.

We define Hoare-Triple like inference rules for the language. Specifications take the form $\{P\}\ c\ \{Q\}$, where P is the precondition, Q is the postcondition, and c is a command. By $\{P\}\ c\ \{Q\}$, we mean that whenever P holds before the execution of command c, then predicate Q holds after the termination of c.

We list basic rules in **Fig. 2**. Here we treat boolean expressions as OOSL assertions, because every boolean expression can be easily mapped to an assertion in OOSL.

Beside basic rules, we have *Frame Rule* that is essential for local reasoning [15].

$$\frac{\{P\}\,c\,\{Q\} \quad FV(R) \cap md(c) = \emptyset}{\{P * R\}\,c\,\{Q * R\}} \qquad \text{(FRAME)}$$

where $FV(R)$ is the set of all program variables in R, and $md(c)$ denotes the variable set modified by c with the following definition:

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

In this paper, we only define the local rules. We can define global rules and backwards rules, as in [15]. For example, here is the backwards reasoning rule for mutation:

$$\{(v = r_1) \wedge (e = r_2) \wedge (r_1.a \mapsto - * (r_1.a \mapsto r_2 \mathrel{-\!\!*} P))\}\ v.a = e\ \{P\} \quad \text{(ASN-II BACK)}$$

Based on Rule (CONS), (FRAME) and Lemma 7, it is easy to prove that this rule is equivalent to Rule (ASN-II).

We use a little example to end this section. This example illustrates that two newly created empty objects are different. As mentioned above, $\mathbf{Object}$ has no attributes, by

$\{P\}\,\mathbf{skip}\,\{P\}$ (SKIP)

$\{P[e/x]\}\; x := e\; \{P\}$ (ASN-I)

$\{x = r_1 \wedge e = r_2 \wedge r_1.a \mapsto -\}\; x.a := e\; \{x = r_1 \wedge e = r_2 \wedge r_1.a \mapsto r_2\}$ (ASN-II)

$\{x_2 = r_1 \wedge r_1.a \mapsto r_2\}\; x_1 := x_2.a\; \{x_1 = r_2 \wedge x_2 = r_1 \wedge r_1.a \mapsto r_2\}$ (ASN-III)

$\{\mathbf{emp}\}\; x := \mathbf{new}\; C()\; \{\exists r \cdot x = r \wedge \mathsf{obj}(r, C)\}$ (NEW)

$$\frac{\{P\}\, c_1\, \{Q\}, \quad \{Q\}\, c_2\, \{R\}}{\{P\}\, c_1; c_2\, \{R\}} \quad \text{(SEQ)}$$

$$\frac{\{b \wedge P\}\, c_1\, \{Q\}, \quad \{\neg b \wedge P\}\, c_2\, \{Q\}}{\{P\}\, \mathbf{if}\;\; b\, c_1\, \mathbf{else}\;\; c_2\, \{Q\}} \quad \text{(COND)}$$

$$\frac{\{b \wedge I\}\, c\, \{I\}}{\{I\}\, \mathbf{while}\, b\, c\, \{\neg b \wedge I\}} \quad \text{(ITER)}$$

$$\frac{P \Rightarrow P', \quad \{P'\}\, c\, \{Q'\}, \quad Q' \Rightarrow Q}{\{P\}\, c\, \{Q\}} \quad \text{(CONS)}$$

$$\frac{\{P\}\, c\, \{Q\}}{\{\exists r \cdot P\}\, c\, \{\exists r \cdot Q\}} \quad r \text{ is free in } P \text{ and } Q \quad \text{(EX)}$$

Fig. 2. Inference Rules for the OO language

Rule (NEW), (FRAME) and Lemma 6,7, we have the following deduction:

$\{\mathbf{true}\}$
$\{\mathbf{emp} * \mathbf{true}\}$
$x \;:=\; \mathbf{new}\; \mathbf{Object}();$
$\{\exists r \cdot x = r \wedge \mathsf{obj}(r, \mathbf{Object}) * \mathbf{true}\}$
$y \;:=\; \mathbf{new}\; \mathbf{Object}();$
$\{\exists r_1, r_2 \cdot x = r_1 \wedge y = r_2 \wedge \mathsf{obj}(r_1, \mathbf{Object}) * \mathsf{obj}(r_2, \mathbf{Object}) * \mathbf{true}\}$
$\{x \neq y\}$

This example shows that OOSL's accurate treatment for empty objects.

5 A Case Study

In this section, we take Schorr-Waite Marking (SWM) Algorithm as an example to show how the specification and verification can be carried on with our logic and inference rules. **Fig. 3** gives an implementation of SWM algorithm in our language. Class $Node$ is the graph node class which has four attributes: $left$ and $right$ are links to the left and right subnodes respectively, flag $mark$ indicates if the node is marked, and flag $check$ is used internally to indicate if its left part has been visited.

To verify the program, we must specify that given any unmarked graph pointed by $root$, after the execution $schorr\_waite$, all nodes in the graph are marked and the graph structure is preserved. Complete verification for these two properties is complicated, especially for the second property, for which we must introduce some mathematical concepts for graphs. Yang [16] presented the first work on verifying SWM with Separation

```
class $Node$ {
  $Node\ left,\ right;$
  Bool $mark,\ check;$ $/*$ $whether\ left\ subtree\ has\ been\ visited$ $*/$
}

void $schorr\_waite(Node\ root)$ {
  $Node\ t,\ p,\ q,\ s;$
  $t := root;\ p :=$ null;
  while $(p \neq$ null $\vee\ (t \neq$ null $\wedge \neg t.mark))$ {
    if $(t =$ null $\vee\ t.mark)$ {
      if $(p.check)$ { $/*$ $pop$ $*/$
        $q := t;\ t := p;\ p := p.right;\ t.right := q;$
      }
      else { $/*$ $swing$ $*/$
        $q := t;\ t := p.right;\ s := p.left;\ p.right := s;\ p.left := q;\ p.check :=$ true;
      }
    }
    else { $/*$ $push$ $*/$
      $q := p;\ p := t;\ t := t.left;\ p.left := q;\ p.mark :=$ true; $p.check :=$ false;
    }
  }
}
```

Fig. 3. Implementation of Schorr-Waite Marking Algorithm

$$
\begin{array}{ll}
\mathsf{node}(r, r_1, r_2, c, m) & \doteq r.left \mapsto r_1 * r.right \mapsto r_2 * r.check \mapsto c * r.mark \mapsto m \\
\mathsf{mtree}(r) & \doteq (r = \mathsf{rnull} \wedge \mathbf{emp}) \vee \\
 & \quad (\exists r_1, r_2 \cdot \mathsf{node}(r, r_1, r_2, -, \mathsf{rtrue}) * \mathsf{mtree}(r_1) * \mathsf{mtree}(r_2)) \\
\mathsf{utree}(r) & \doteq (r = \mathsf{rnull} \wedge \mathbf{emp}) \vee \\
 & \quad (\exists r_1, r_2 \cdot \mathsf{node}(r, r_1, r_2, -, \mathsf{rfalse}) * \mathsf{utree}(r_1) * \mathsf{utree}(r_2)) \\
\mathsf{sbot}(r) & \doteq \exists r_1, r_2, c \cdot \mathsf{node}(r_b, r_1, r_2, c, \mathsf{rtrue}) * \\
 & \qquad ((c = \mathsf{rtrue} \wedge r_2 = \mathsf{rnull} \wedge \mathsf{mtree}(r_1)) \vee \\
 & \qquad\ (c = \mathsf{rfalse} \wedge r_1 = \mathsf{rnull} \wedge \mathsf{utree}(r_2))) \\
\mathsf{sseg}(r_t, r_b) & \doteq (r_t = r_b \wedge \mathsf{sbot}(r_b)) \vee (\exists r_1, r_2, c \cdot \mathsf{node}(r, r_1, r_2, c, \mathsf{rtrue}) * \\
 & \qquad ((c = \mathsf{rtrue} \wedge \mathsf{mtree}(r_1) * \mathsf{sseg}(r_2, r_b)) \vee \\
 & \qquad\ (c = \mathsf{rfalse} \wedge \mathsf{utree}(r_2) * \mathsf{sseg}(r_1, r_b)))))
\end{array}
$$

Fig. 4. User-defined assertions for Schorr-Waite Algorithm

Logic, where he gave a complete verification of SWM on binary tree. For the verification, he introduced some auxiliary mathematical concepts, including tree and list. As an illustration possibly been included in the paper, here, we do not focus on a complete verification. We simplify the specification in two aspects: We require the input is a tree, and we do not care about the tree structure preservation. So we take a specification here as: given any unmarked tree, after the execution of the program, all nodes in the tree are marked. Though this specification is not complete, it is a good example to illustrate the usefulness and power of OOSL.

At first, we introduce some user-defined assertions, as shown in **Fig. 4**. Assertion node specifies a single tree node; $\mathsf{mtree}(r)$ and $\mathsf{utree}(r)$ specify that the whole tree from r is marked or unmarked, respectively. Assertions sbot and sseg talk about the implicit stack and the segment of nodes reachable through the stack. In details, $\mathsf{sbot}(r)$ specifies that r is the only node in the stack and has been marked; and if the flag $check$ of this node is true, then its left subtree is marked and its right subtree is null, otherwise, its left subtree is null and right subtree is unmarked. On the other hand, $\mathsf{sseg}(r_t, r_b)$ specifies a stack with r_t as its top element and r_b its bottom element. Further, if $r_t = r_b$, then the stack has only one node r_b; otherwise, every node in the stack has been visited, and if the $check$ flag of a node is true, then its left subtree is marked and its $right$ field records the next node in the stack, otherwise its right subtree is unmarked and its $left$ field records the next node in the stack.

Now we give a specification for the SWM program:

$$\{root = r_{root} \wedge \mathsf{utree}(r_{root})\}\ SWM\ \{root = r_{root} \wedge \mathsf{mtree}(r_{root})\} \tag{1}$$

Here SWM represents the body of the function $schorr\_waite$ shown in **Fig. 3**.

For proving the specification, the key-point is defining a suitable loop invariant. We define the loop invariant I as follows (with auxiliaries Inv_p and Inv_r):

$$\begin{array}{ll} I & \doteq \exists r_t, r_p \cdot t = r_t \wedge p = r_p \wedge (r_p = \mathsf{rnull} \Rightarrow r_t = r_{root}) \wedge I_p(r_p) * I_t(r_t) \\ I_p(r_p) & \doteq (r_p = \mathsf{rnull} \wedge \mathbf{emp}) \vee (r_p \neq \mathsf{rnull} \wedge \mathsf{sseg}(r_p, r_{root})) \\ I_t(r_t) & \doteq \mathsf{mtree}(r_t) \vee \mathsf{utree}(r_t) \end{array}$$

This loop invariant says:

- If p is null, which means the stack is empty, then the value of t must be $root$;
- The whole Opool consists of two separated parts. The first part, that is specified by $I_p(r_p)$, is the part of nodes reachable from the implicit stack with p referring to its top element. If p is null then this part is empty. The second part, that is specified by $I_p(r_t)$, is a tree denoted by t. The nodes in the tree must be all marked or unmarked.

We can simply prove the following facts:

The precondition establishes the loop invariant:

$$\begin{array}{l} \{root = r_{root} \wedge \mathsf{utree}(r_{root})\} \\ t\ :=\ root;\ p\ :=\ \mathbf{null}; \\ \{root = r_{root} \wedge t = r_{root} \wedge p = \mathsf{rnull} \wedge \mathsf{utree}(r_{root})\} \\ \{I\} \end{array}$$

The postcondition holds when the loop ends:

$$\begin{array}{l} \quad(\exists r_p, r_t \cdot p = r_p \wedge t = r_t \wedge r_p = \mathsf{rnull} \wedge (r_t = \mathsf{rnull} \vee r_t.mark \hookrightarrow \mathsf{rtrue})) \wedge I \\ \Rightarrow t = r_{root} \wedge \mathsf{mtree}(r_{root}) \end{array}$$

Now we prove that the loop invariant is preserved by the loop body. The whole proof is split into three cases according to the conditional branches, and all necessary lemmas and rules used in the deduction can be found in Section 3.3 and 4.

We put here only the case for the branch *Pop*. The other two cases are given in the Appendix.

Case *Pop* : The condition is $p \neq \mathbf{null} \wedge (t = \mathbf{null} \vee t.mark) \wedge p.check$. We have the following deduction:

$$
\begin{array}{l}
\{(\exists r_t, r_p \cdot t = r_t \wedge p = r_p \wedge r_p \neq \mathsf{rnull} \wedge \\
\quad (r_t = \mathsf{rnull} \vee r_t.mark \hookrightarrow \mathsf{rtrue}) \wedge r_p.check \hookrightarrow \mathsf{rtrue}) \wedge I\} \\
\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot t = r_t \wedge p = r_p \wedge \\
\quad (\mathsf{mtree}(r_t) * \\
\quad\; ((r_p = r_{root} \wedge r_{pr} = \mathsf{rnull} \wedge \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_{pl})) \vee \\
\quad\;\; (\mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_{pl}) * \mathsf{sseg}(r_{pr}, r_{root}))))\} \\
\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot t = r_t \wedge p = r_p \wedge \\
\quad (\mathsf{mtree}(r_t) * \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rtrue}, \mathsf{rtrue}) * \\
\quad\; \mathsf{mtree}(r_{pl}) * ((r_p = r_{root} \wedge r_{pr} = \mathsf{rnull} \wedge \mathbf{emp}) \vee \mathsf{sseg}(r_{pr}, r_{root}))))\} \\
q := t;\ t := p;\ p := p.right; \\
\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot q = r_t \wedge t = r_p \wedge p = r_{pr} \wedge \\
\quad (\mathsf{mtree}(r_t) * \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_{pl}) * \\
\quad\; ((r_p = r_{root} \wedge r_{pr} = \mathsf{rnull} \wedge \mathbf{emp}) \vee \mathsf{sseg}(r_{pr}, r_{root})))\} \\
t.right := q; \\
\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot q = r_t \wedge t = r_p \wedge p = r_{pr} \wedge \\
\quad (\mathsf{mtree}(r_t) * \mathsf{node}(r_p, r_{pl}, r_t, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_{pl}) * \\
\quad\; ((r_p = r_{root} \wedge r_{pr} = \mathsf{rnull} \wedge \mathbf{emp}) \vee \mathsf{sseg}(r_{pr}, r_{root})))\} \\
\{\exists r_p, r_{pr} \cdot t = r_p \wedge p = r_{pr} \wedge (r_{pr} = \mathsf{rnull} \Rightarrow r_p = r_{root}) \wedge \\
\quad (\mathsf{mtree}(r_p) * ((r_{pr} = \mathsf{rnull} \wedge \mathbf{emp}) \vee \mathsf{sseg}(r_{pr}, r_{root})))\} \\
\{I\}
\end{array}
$$

With the proofs for the other two cases together, we conclude that the specification (1) holds. The proof for the full functional specification of Schorr-Waite Marking Algorithm will be one of our future works.

6 Related Work and Conclusion

To develop a full-armed logic framework for the specification and verification of OO programs is a long standing research goal in the software area. The work presented here is an attempt in this direction. As the last part of the paper, we overview some closely related work, make some comparisons, and list some future works.

Middelkoop tried to extend Separation Logic to OO domain in [9], where only the storage model is revised, and the assertion language remains. In their work, the separation conjunction operator $*$ is defined on the object level, but not on the attribute level; hence an object cannot be split. In this case, the atomic unit is a whole object, that limits the power of Frame Rule considerably.

Parkinson developed a revised Separation Logic for OO programs in his thesis [14] and some other papers. Although the start points are similar to ours, the framework is very different. In Parkinson's work, the program states are defined as:

$$
\begin{array}{lll}
\mathrm{Heaps} & \widehat{=} & (\mathrm{OIDS} \times \mathrm{FieldNames} \rightharpoonup_{\mathsf{fin}} \mathrm{Values}) \times (\mathrm{OIDS} \rightharpoonup_{\mathsf{fin}} \mathrm{Class}) \\
\mathrm{Stacks} & \widehat{=} & \mathrm{Vars} \to \mathrm{Values} \qquad \mathrm{Interpretations} \ \widehat{=}\ \mathrm{AuxVars} \to \mathrm{Values} \\
\mathrm{States} & \widehat{=} & \mathrm{Heaps} \times \mathrm{Stacks} \times \mathrm{Interpretations} \qquad .
\end{array}
$$

The first part of a heap $h \in \mathrm{Heaps}$ stores values of objects' attributes, and the second part stores their type information. An object is not explicit, but only a set of cells with the same id from OIDS. The additional component "$\mathrm{Interpretations}$" records values of logical variables. Taking this $\mathrm{Interpretations}$ into program states looks not nice, because it has no correspondent in practice. Clearly, logical variables are used only in verification, but not in execution. It is not nature to take them as a specific and independent part in program states. In this logic, operator $*$ separates only the first part of Heaps in states, thus different empty objects can not be separated. As seen, our state model is different, which records only information of program variables and objects. We have a novel definition for the separation of heaps (Opools), that makes it possible to separate the heaps efficiently even they contain empty objects.

Additionally, the logic in [14] adopts intuitionistic semantics, thus assertions preserve true with heap extension. This makes it impossible to express precise specifications about heaps, e.g., the simplest statement "current heap is empty". Consequently, no precise property about OO programs can be proved in this framework. Our logic takes the classical semantics, thus is more expressive [5]. The precise assertions are default, and intuitionistic assertions can also be written (ref. to [15]).

Parkinson *et al.* [13] developed a framework and some techniques based on their logic, for verifying OO programs modularly. We can also develop similar framework within our OOSL, which is our current work.

From these analysis and precognition, we make our choices. We take the reference model for OO languages, the assertion language based on Separation Logic, and the logic with classical semantics. Of course, what reported here is only a preliminary work.

In this paper, we present a state model for OO programs, and a novel definition for the separation of object heaps. Based on the storage model, we define an OO Separation Logic with some new assertion forms. We give a full treatment on user-defined basic assertions and introduce the concept of logic environment into our framework. We list the necessary conditions which guarantee the existence of the fixed point for a logic environment. We define semantics for the logic and prove some properties (reasoning rules) for it. We introduce a simple OO language with a set of inference rules based on the logic. The Schorr-Waite Marking algorithm is used as the example to illustrate how the specification and verification can be done here.

As for the future work, first, it would be interesting to study properties of OO Separation Logic, provide and prove more inference rules, in order to pave the way for more effective reasoning for OO programs. We also take interests in the connection between OOSL and the Separation Logic, as mentioned in Section 3.2.

Second, it is important to extend the language used here to support all higher level OO features, e.g., class declaration, method definition and invocation, inheritance, etc. Further, we should try to develop more powerful framework to do modular specification and verification like techniques in [11, 13].

Third, accounting to the procedural paradigm, Weakest Precondition (WP) semantics plays a central role in semantics studies, and the foundation stone for many theoretical work deeply related to the software engineering, including specification, verification, refinement, programming from specifications [10], specification-based code generation, etc. A well-defined WP semantics might play similar role in OO paradigm.

We will try to develop a WP Semantics for an OO language with all important OO features, which could enables us to define data refinement and program refinement. With WP semantics as a base, we could study program transformation, and the refinement relationship between programs/specifications at different abstract levels, therefore provide the possibility of programming from specifications or code generation.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, LNCS 3362, pages 49–69. Springer, 2005.
2. Marius Bozga, Radu Losif, and Yassine Lakhnech. On logics of aliasing. *SAS 2004*, LNCS 3148:pp.344–360, 2004.
3. Yifeng Chen and J W Sanders. A pointer logic for object diagrams. Technical report, International Institute for Software Technology, The United Nations University, 2007.
4. C.A.R. Hoare and Jifeng He. A trace model for pointers and objects. *ECOOP'99, Object Oriented Programming*, 1628/1999:pp.344–360, 1999.
5. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*. ACM, 2001.
6. Wei Ke, Zhiming Liu, Shuling Wang, and Liang Zhao. A graph-based operational semantics of OO programs. In *ICFEM 2009*, volume 5885 of *LNCS*, pages 347–366. Springer, 2009.
7. G. T. Leavens, A. L.Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
8. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.
9. Ronald Middelkoop, Kees Huizing, and Ruurd Kuiper. A separation logic proof system for a class-based language. In *Proceedings of the Workshop on Logics for Resources, Processes and Programs (LRPP)*, 2004.
10. C. Morgan. *Programming from Specifications*. Prentice Hall, 1998.
11. P. Müller. *Modular specification and verification of object-oriented programs*. Springer-Verlag, LNCS 2262, 2002.
12. James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke. Towards a model of encapsulation. Technical report, Elvis Software Design Research Group, 2003.
13. Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Principles of Programming Languages (POPL'08)*. ACM, 2008.
14. Matthew John Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, 2005.
15. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*. IEEE Computer Society, 2002. Invited paper.
16. Hongseok Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001. (Technical Report UIUCDCS-R-2001-2227).
17. Liang Zhao, Xiaojian Liu, Zhiming Liu, and Zongyan Qiu. Graph transformations for object-oriented refinement. *Formal Aspects in Computing*, 21(1):103–131, 2009.

Appendix

We include here proofs for the other two cases for proving that the program implementing Schorr-Waite Marking algorithm, given in **Fig. 3**, meets specification (1).

The first case, for *Swing*, is little complicated, as the case *Pop*. However, the deduction is also straightforward. The last case *Push* is much simpler.

Case Swing: $p \neq \mathbf{null} \wedge (t = \mathbf{null} \vee t.mark) \wedge \neg p.check$

$\{(\exists r_t, r_p \cdot p = r_p \wedge t = r_t \wedge r_p \neq \mathsf{rnull} \wedge$
$\quad (r_t = \mathsf{rnull} \vee r_t.mark \hookrightarrow \mathsf{rtrue}) \wedge r_p.check \hookrightarrow \mathsf{rfalse}) \wedge I\}$
$\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot p = r_p \wedge t = r_t \wedge$
$\quad (\mathsf{mtree}(r_t) *$
$\quad\quad ((r_p = r_{root} \wedge r_{pl} = \mathsf{rnull} \wedge \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rfalse}, \mathsf{rtrue}) * \mathsf{utree}(r_{pr})) \vee$
$\quad\quad (\mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rfalse}, \mathsf{rtrue}) * \mathsf{utree}(r_{pr}) * \mathsf{sseg}(r_{pl}, r_{root}))))\}$
$\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot p = r_p \wedge t = r_t \wedge$
$\quad (\mathsf{utree}(r_{pr}) *$
$\quad\quad ((r_p = r_{root} \wedge r_{pl} = \mathsf{rnull} \wedge \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rfalse}, \mathsf{rtrue}) * \mathsf{mtree}(r_t)) \vee$
$\quad\quad (\mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rfalse}, \mathsf{rtrue}) * \mathsf{mtree}(r_t) * \mathsf{sseg}(r_{pl}, r_{root}))))\}$
$q := t;\ t := p.right;\ s := p.left;$
$\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot q = r_t \wedge p = r_p \wedge t = r_{pr} \wedge s = r_{pl} \wedge$
$\quad (\mathsf{utree}(r_{pr}) *$
$\quad\quad ((r_p = r_{root} \wedge r_{pl} = \mathsf{rnull} \wedge \mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rfalse}, \mathsf{rtrue}) * \mathsf{mtree}(r_t)) \vee$
$\quad\quad (\mathsf{node}(r_p, r_{pl}, r_{pr}, \mathsf{rfalse}, \mathsf{rtrue}) * \mathsf{mtree}(r_t) * \mathsf{sseg}(r_{pl}, r_{root}))))\}$
$p.right := s;\ p.left := q;\ p.check := \mathbf{true};$
$\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot q = r_t \wedge p = r_p \wedge t = r_{pr} \wedge s = r_{pl} \wedge$
$\quad (\mathsf{utree}(r_{pr}) *$
$\quad\quad ((r_p = r_{root} \wedge r_{pl} = \mathsf{rnull} \wedge \mathsf{node}(r_p, r_t, r_{pl}, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_t)) \vee$
$\quad\quad (\mathsf{node}(r_p, r_t, r_{pl}, \mathsf{rtrue}, \mathsf{rtrue}) * \mathsf{mtree}(r_t) * \mathsf{sseg}(r_{pl}, r_{root}))))\}$
$\{\exists r_t, r_p, r_{pl}, r_{pr} \cdot p = r_p \wedge t = r_{pr} \wedge (\mathsf{utree}(r_{pr}) * \mathsf{sseg}(r_p, r_{root}))\}$
$\{I\}$

Case Push: $t \neq \mathbf{null} \wedge \neg t.mark$

$\{(\exists r_t \cdot t = r_t \wedge r_t \neq \mathsf{rnull} \wedge r_t.mark \hookrightarrow \mathsf{rfalse}) \wedge I\}$
$\{\exists r_t, r_p, r_{tl}, r_{tr} \cdot t = r_t \wedge p = r_p \wedge (r_p = \mathsf{rnull} \Rightarrow r_t = r_{root}) \wedge$
$\quad (I_p(r_p) * \mathsf{node}(r_t, r_{tl}, r_{tr}, -, \mathsf{rfalse}) * \mathsf{utree}(r_{tl}) * \mathsf{utree}(r_{tr}))\}$
$q := p;\ p := t; t := t.left;$
$\{\exists r_t, r_p, r_{tl}, r_{tr} \cdot q = r_p \wedge p = r_t \wedge t = r_{tl} \wedge (r_p = \mathsf{rnull} \Rightarrow r_t = r_{root}) \wedge$
$\quad (I_p(r_p) * \mathsf{node}(r_t, r_{tl}, r_{tr}, -, \mathsf{rfalse}) * \mathsf{utree}(r_{tl}) * \mathsf{utree}(r_{tr}))\}$
$p.left := q;\ p.mark := \mathbf{true}; p.check := \mathbf{false};$
$\{\exists r_t, r_p, r_{tl}, r_{tr} \cdot q = r_p \wedge p = r_t \wedge t = r_{tl} \wedge (r_p = \mathsf{rnull} \Rightarrow r_t = r_{root}) \wedge$
$\quad (I_p(r_p) * \mathsf{node}(r_t, r_p, r_{tr}, \mathsf{rfalse}, \mathsf{rtrue}) * \mathsf{utree}(r_{tl}) * \mathsf{utree}(r_{tr}))\}$
$\{\exists r_t, r_p, r_{tl}, r_{tr} \cdot q = r_p \wedge p = r_t \wedge t = r_{tl} \wedge \mathsf{utree}(r_{tl}) *$
$\quad ((r_p = \mathsf{rnull} \wedge r_t = r_{root} \wedge \mathbf{emp}) \vee \mathsf{sseg}(r_p, r_{root})) *$
$\quad\ \mathsf{node}(r_t, r_p, r_{tr}, \mathsf{rfalse}, \mathsf{rtrue}) * \mathsf{utree}(r_{tr})\}$
$\{\exists r_t, r_p, r_{tl}, r_{tr} \cdot p = r_t \wedge t = r_{tl} \wedge (\mathsf{utree}(r_{tl}) * \mathsf{sseg}(r_t, r_{root}))\}$
$\{I\}$