OO Separation Logic and Specification/Verification of OO Programs[1]

Liu Yijing and Qiu Zongyan

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

Abstract

With a general storage model that reflects features of object oriented (OO) languages with pure reference semantics, we develop an OO Separation Logic (OOSL) for specifying and verifying OO programs. We define the semantics for OOSL, and show that it has essential and useful properties for OO reasoning. We then investigate some important problems related to specification and verification of OO programs, and develop an OO language with specification features and a bundle of verification rules for reasoning programs. We define the syntactic and scope rules, as well as the inheritance and overriding rules for the specification facilities, and an important concept named *specification predicate* to connect abstract specification with implementation details. We use a set of typical OO programs to show the potential and power of our framework, how the specification facility supports abstractions, and how the specification and verification can be done in a modular manner.

Keywords: Specification, Verification, OO, Separation Logic

1. Introduction

Object Orientation (OO) is a mainstream paradigm in software development. For increasingly demand on reliability and correctness recently, the requirement for powerful and useful techniques for specifying/verifying OO programs become even urgent. Generally speaking, 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.

Core OO features, saying modularity, encapsulation, inheritance, polymorphism, etc. are extremely important in the development of complex systems. Their suitable application can enhance scalability of systems greatly. However, encapsulation and modularity imply information hiding and invisible implementation details, and polymorphism enables dynamic behaviors. They bring also great challenges to verification, because verification is a static procedure, which needs inherently knowledge of implementation details.

The first formal concept to deal with inheritance and polymorphism is *behavioral subtyping* [20], which has become a useful guide for good OO programming, as well as for specification and verification. An OO program with behavioral subtyping feature gives more support to reason its behavior statically and modularly. And abstraction is the other key to modular verification of OO programs, many concepts have been proposed, e.g., *model field*/*abstract field* [9, 18, 22], *data group* [19], *abstract predicate family* [24, 25], and *pure methods* [28].

Many early works on OO verification targeted programs with class structures but carrying computations of numerical values, for example [16, 3]. People investigated many OO issues in this circumstance, and brought to light many interesting problems related to data abstraction, class inheritance, etc. However, working only up to this level is not enough. For verifying more wide spectrum of OO programs, we must take the mutable object structures into account. For this ambitious goal, many new issues need to be (re)considered and conquered.

As the first step, we need a memory model to abstract mutable object structures properly. For reasoning OO programs, we need a logic to describe the states of OO programs and the effects made by commands on the states. Recently, many people think that *Separation Logic* [27] (SL for short) is superior here, either used directly [21], or adjusted [24].

However, as pointed by Parkinson [25], the early works do not address the inheritance in a satisfactory manner. They either restricts behaviors of subclasses, or requires re-verification of inherited methods. To solve the problem, Parkinson [25] developed a framework, while Chin [10] had a similar idea in the same time. Both suggested to use dual specifications for each method, to avoid re-verification in the present of inheritance.

In this report, we present a wide-range study on the specification/verification of OO programs, a developed OO memory model, a revised separation logic for describing states and transitions of OO program, a gradually developed verification framework, a set of working examples which show many typical and important specification and verification issues, as well as the way our framework addresses them.

In the first part of this report, we build the theoretical foundation for specification and verification of OO programs. We design a special separation logic OOSL (for OO Separation Logic) for describing program states and properties of the states, i.e., the sets of the states. We develop a classical semantics for the logic, but not the intuitionistic semantics. This make the logic possible to describe precise properties of OO states. In addition, we design a model OO language, named μJava, which captures most important OO features, and give it a weakest precondition semantics (WP semantics) using the OOSL. We have proved that the semantics is both sound and complete, with respect to an operational semantics for μJava. This part of the work gives us a solid foundation for carrying on the work for specification and verification of OO programs.

As the second part of this report, we focus on program verification related issues in the OO world. We define first the concepts of method specifications and correctness of methods, then the refinement relation between specifications, all based on the WP semantics. Then we go into a deep study on the specification and verification of OO programs, taking μJava programs in the work. We develop gradually a set of Hoare-style inference rules for reasoning OO programs, and obtain at last a framework which supports modular specification and verification, in the present of information hiding of classes, as well as method inheritance and overriding. One fundamental is to support the abstract-level specification, and use *specification predicates* to connect the abstract specification with the implementation details.

To give an intuition for these, we illustrate them by a simple example. Fig. 1 gives a typical piece of OO code, where class *Node* defines nodes holding boolean values; *Queue* defines a kind of simple queues, where field *hd* holds a linked list of *Node* objects with a head node, and the node *hd.nxt* holds the first value in the queue. Method *enqueue* inserts a value into the queue. *EQueue* is a subclass of *Queue* which defines *faster queue*s. A new field *tl* in *EQueue* object points to the last node of its list, and a new *enqueue* overrides the old one in *Queue*.

To specify and verify a program as this one, we need to consider some important issues. Most of them are general in the specification and verification of OO programs:

```
class Node : Object {
  public Bool val; public Node nxt;
  Node(Bool b) { this.val = b; this.nxt = null; }
}
class Queue : Object {
  Node hd;
  Queue(){Node x; x = new Node(false); this.hd = x;}
  void enqueue(Bool b) {
    Node p, q, n; p = this.hd; q = p.nxt;
    while (q!=null){ p = q; q = p.nxt;}
    n = new Node(b); p.nxt = n;
  }
}
```

```
class EQueue : Queue {
  Node tl;
  EQueue() {
    Node x;
    x = new Node(false);
    this.hd = x; this.tl = x;
  }
  void enqueue(Bool b) {
    Node p, n;
    p = this.tl; n = new Node(b);
    p.nxt = n; this.tl = n;
  }
}
```

Figure 1: Example OO code: *Queue* and *EQueue*

- In OO field, developers often distinguish interfaces from implementations, to prevent close dependence on implementation details, achieve high degree of modularity, and organize inheritance hierarchy. These ideas may also be important in formal specification and verification. As an example, for method *enqueue*, we may sometimes better say abstractly its effect on a sequence of values in the queue only, but nothing about the linked list.

- Having abstract specifications, we need a way to link them with code. Clearly, the link should involve implementation details in general, but as in programming, we may want to prevent details from leaking out. For example, for *Queue*, we need to link the abstract queue concept to the concrete linked lists.

- Issues above are general to programs with data abstraction. For OO, we must consider inheritance and overriding. Having *Queue* well-specified and verified, when we introduce subclass *EQueue*, it is desired that existing specifications and verifications can be reused, because *EQueue* is behavior subtype of *Queue*. However, now we have a very different implementation. On one side, in sharing a same specification, it means that the same assertion should have different meaning, thus comes the polymorphic specification. On the other side, we need to connect the specification with the new implementation. In addition, the inference rules must support these.

Our specification and verification framework embodies above ideas. A fully specified program of **Fig. 1** using our notation is given in **Fig. 16**. In fact, by a set of syntactic and semantic rules, our framework introduce polymorphism into the specification and verification world. Due to our limited knowledge, this is the first work which makes this idea clear and explicit.

The main contributions of the work include:

- We give an abstract memory model to capture important characters of object structures of OO programs, and a new definition of separation for expressing both field-level and object-level problems. Empty objects can be naturally represented and reasoned.

- We develop a revised separation logic, OO Separation Logic (OOSL), for expressing OO program states and properties. 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. OOSL adopts classical semantics, then is more expressive than the logic with

intuitionistic semantics. This makes the logic capable to specify precise properties of OO programs. Properties of OOSL are explored, especially *separated assertions*, that is very useful in reasoning OO programs.

- We develop a verification framework which supports *abstract specification* for information hiding and encapsulation. We propose a concept *specification predicate* to connect abstract specification with implementation details within a class. To support modularity, we integrate specification facilities with important OO features, and define their scope rules (visibility), as well as rules for their inheritance, overriding, etc. Based on this mechanisms, we show only one specification for each method is enough to support modular verification, and avoid re-verification in the present of inheritance.

- By embedding the verification framework into a small OO language, we show how information hiding, inheritance, overriding, etc. can be integrated with specification features to enhance verification abilities for OO programs, and illustrate that these concepts are important not only in programming, but also in specifying and verifying OO programs. We define a bundle of Hoare-style rules for generating proof obligations.

- Examples are developed to illustrate, in our framework, how the specification and verification can be carried out modularly, and how the gaps between a verification logic and implementation details are bridged smoothly.

On most notable point is, our work shows that the polymorphism is very important and useful in the specification and verification world. Based on this novel idea, we illustrate that abstract specifications, specification predicates and relative rules form a solid base for modular specification and verification of OO programs.

The rest of this report is organized as follows: We introduce our memory model for OO programs in **Section 2**, and then the object oriented separation logic (OOSL) in **Section 3**. We define a small OO language μJava in **Section 4**, as well as its WP semantics with a discussion on properties of this semantics, especially the soundness and completeness theorems. Then we try to develop a modular specification and verification framework for μJava. **Section 5** introduce the concepts and notations of method specification and refinement. **Section 6** defines a basic verification framework, which is powerful enough to do body verification but not ready for class inheritance. We develop abstract and polymorphic specification techniques in **Section 7** and show these techniques really support the modular verification. In **Section 8**, we investigate the concept *ad hoc inheritance* carefully and add explicit code reuse to enrich the semantics of class inheritance. At last, we discuss some related work and conclude.

2. An OO Storage Model

Now we introduce a storage model for OO programs with pure reference nature. It is similar to the classical Stack-Heap model. We assume three basic sets Name, Type and Ref, where

- Name: an infinite set of names, used for naming constants, variables, fields, etc. Three special names, `true`, `false`, `null` $\in$ Name, denote boolean and null constants.

- Type: an infinite set of types, including predefined types and user-defined types (*classes*). We use $T_1 <: T_2$ to state that T_1 is a subtype of T_2, perhaps they are the same. We assume three predefined types `Object`, `Null` and `Bool`, where `Object` is the super type of all

classes, `Null` is the subtype of all classes, and `Bool` is the type of boolean objects. Given a type, we can obtain its field-type relation by function $\mathsf{fields} : \mathsf{Type} \to \mathsf{Name} \to \mathsf{Type}$; and we define $\mathsf{fields}(\texttt{Object}) = \mathsf{fields}(\texttt{Null}) = \mathsf{fields}(\texttt{Bool}) = \emptyset$. Other predefined types, such as Integer, can be added easily, but we consider only boolean type here.

- Ref: an infinite set which are the identities of objects. Corresponding to Name constants, Ref contains three basic references rtrue, rfalse, and rnull, where rtrue, rfalse refer to `Bool` objects, and rnull never refers to any object. We assume two primitive functions for Ref:[2] For any $r_1, r_2 \in \mathsf{Ref}$, $\mathsf{eqref}(r_1, r_2)$ = true iff r_1 is same to r_2. $\mathsf{type} : \mathsf{Ref} \to \mathsf{Type}$, gives the type of object referred by reference in the running time. We define $\mathsf{type}(\mathsf{rtrue}) = \mathsf{type}(\mathsf{rfalse}) = \texttt{Bool}$, and $\mathsf{type}(\mathsf{rnull}) = \texttt{Null}$.

We assume a function dtype, where $\mathsf{dtype}(v)$ gives the declaration type of constant or variable v. In fact, Name, Type and functions (relations) defined on them, such as type, dtype, <: and fields, are the basic facilities for describing states of OO programs. They gives the static information of programs, especially the type information.

We define the storage model as (here "$\rightharpoonup_{\mathsf{fin}}$" denotes finite partial functions.):

$$\mathsf{Store} \mathrel{\hat{=}} \mathsf{Name} \rightharpoonup_{\mathsf{fin}} \mathsf{Ref} \qquad \mathsf{Opool} \mathrel{\hat{=}} \mathsf{Ref} \rightharpoonup_{\mathsf{fin}} \mathsf{Name} \rightharpoonup_{\mathsf{fin}} \mathsf{Ref} \qquad \mathsf{State} \mathrel{\hat{=}} \mathsf{Store} \times \mathsf{Opool}$$

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

We will use $r, r_1, \ldots$ to denote references, and $a, a_1, \ldots$ fields 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 fields of o to their values (also references). For the "domain" of O, we sometimes want to say a subset of Ref associated with a set of objects as discussed above. On the other hand, we sometimes want to say a subset of $\mathsf{Ref} \times \mathsf{Name}$ associated with a set of values. We will use $\mathsf{dom}\, O$ for the first case, and define $\mathsf{dom}_2\, O \mathrel{\hat{=}} \{(r, a) \mid r \in \mathsf{dom}\, O, a \in \mathsf{dom}\, O(r)\}$ to get the reference-field pairs of non-empty objects in O.

For the program states to be meaningful, we need some regularity. Now we define the concept that a state is consistent with the static information of the program, i.e., the *well-typed state*.

Definition 1 (Well-typed Store)**.** A store σ is well-typed iff all variables hold valid values. Foramlly

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

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

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

where $\mathsf{Att}(r) \mathrel{\hat{=}} \mathsf{dom}\ \mathsf{fields}(\mathsf{type}(r))$.

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

Note that $\mathsf{fields}(\mathsf{type}(r))$ is a function from field set (i.e., the set of an object to which r points) to Type. The first condition requires that each field in O is valid according to its object, and holds a value of a correct type. The second condition requires when O contains a non-empty object (according to the type of the object), then it must contains at least one field of the object. Thus we can identify empty objects.

Suppose we have $\mathsf{dom}\,\mathsf{fields}(C) = \{a_1, a_2, a_3\}$, and a program state where $\mathsf{type}(r_1) = \texttt{Object}$, $\mathsf{type}(r_2) = C$. Then is is easy to 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 fields. 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.

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 (i.e., the state of a field of an object) in the Opool. Here "$-$" represents some value which we do not care about in the context.

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

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

where the $\oplus$ on the right hand side is the standard function overriding operator. For example, $O_1 \oplus \{(r, a, r')\}$ gives a new Opool, where only the value of field a of the object pointed by r is modified (to r').

As in Separation Logic, we use $O_1 \perp O_2$ to indicate that Opools O_1 and O_2 are separated from each other. However, the definition for $\perp$ is adapted for separating object pools,

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

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 fields, respectively (the well-typedness also guarantees this); and these two subsets must be disjoint. This means that we can separate fields 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 for the separation takes the basic cell (r, a, r') as an unit, and treats also carefully the empty objects. The separation here is a revision of separation concept in Separation Logic, and takes the characteristics of OO programs into account. 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 fields are references to objects[3]. A special case is that their values can be null to mean referring to no object. This model induces a great possibility of sharing: different variables can share references, so do the fields, and they can have sharing with variables. For reasoning these features, we define an OO Separation Logic (OOSL in short).

3.1. Assertions Language

We use Ψ for the set of all assertions of OOSL, and $\psi, \psi_1, \ldots$ 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 in OOSL are of two kinds, namely *primitive assertions* and *user-defined assertions*. The primitive assertions here have the forms defined by following rules:

$$\alpha \ ::= \ \texttt{true} \mid \texttt{false} \mid v = r \mid r_1 = r_2 \mid r : T \mid r <: T \qquad \beta \ ::= \ \mathbf{emp} \mid r_1.a \mapsto r_2 \mid \mathsf{obj}(r)$$

where v denotes a variable or constant name, r denotes a reference. In fact, here r serves as either "a reference" or "a reference variable" (a logic variable).

As shown, primitive assertions fall into two categories, where

- α denotes the assertions which are independent of the Opools. As said before, we take references as atomic values in the logic. For two references r_1, r_2, $r_1 = r_2$ holds iff $\mathsf{eqref}(r_1, r_2)$. Here $r = v$ denotes the value equality. We treat $r = v$ the same as $v = r$. Assertion forms $r : T$ and $r <: T$ states that reference r refers to an object of exactly type of T, or of some subtype of T respectively.

- β denotes the assertions involving Opools. Empty and singleton assertions take the similar forms as in Separation Logic. As said before, a cell in the Opool is a field-value binding of an object denoted by a reference, thus the singleton assertion takes the form $r_1.a \mapsto r_2$. In addition, assertion $\mathsf{obj}(r)$ indicates that r refers to a complete object of type T, and the Opool contains exactly this object.

To make OOSL clear and simple, we do not define $v.a \mapsto r$ as a primitive assertion, because it can be defined as $\exists r' \cdot v = r' \wedge r.a \mapsto r'$. In Separation Logic, we can use $l \mapsto -$ or $l \hookrightarrow\mapsto -$ to denote that location l is in current heap, in fact, the heap contains exactly this location. Due to the existence of empty object, we cannot use $r.a \mapsto -$ or $r.a \hookrightarrow\mapsto -$ to express that the object which r refers to is in current Opool. To solve this problem, we introduce the special assertion form $\mathsf{obj}(r)$. We will use $\mathsf{obj}(r)$ when we do not care about r's type.

We allow user-defined 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. These definitions are recorded in a *Logic Environment* Λ defined as:

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

[3] One exception might be variables and fields of the primitive types, e.g., the boolean or integer, while many languages use value model for them for efficiency.

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 assertion correlated with $\overline{r}$. Clearly, a logic environment Λ is simply a sequence of predicate definitions. Recursive definitions are allowed. As a well-formed logic environment, Λ must be self-contained, that is: 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*[4], 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.

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

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

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 program texts, thus must be 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 in it with variable or constant v (reference r). 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:

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

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

3.2. Semantics

Now we define a *least fix-point semantics* for OOSL by a semantic function which maps every assertion $\psi \in \Psi$ to a subset of State. For this goal, we first define a semantics for Λ.

We introduce a family of *Predicate Functions*. For any $n \geq 0$, let $\mathcal{P}_n \triangleq \mathsf{Ref}^n \to \mathbb{P}(State)$ be the set of functions from n references to subsets of State. Let $\mathcal{P} \triangleq \bigcup_n \mathcal{P}_n$ be the set of all possible predicate functions. We introduce a function $\mathsf{arity} : \mathcal{P} \to \mathbf{N}$ to extract the arity of the 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 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 $\bot_{\mathcal{P}_n} = \{(r_1, ...r_n) \mapsto \emptyset\}$, $\top_{\mathcal{P}_n} = \{(r_1, ...r_n) \mapsto \text{State}\}$ as its bottom and top elements.

With predicate functions, we have definition:

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:

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

$$
\begin{array}{ll}
\text{[I-FALSE]}\mathcal{M}_{\mathcal{I}}(\texttt{false}) = \emptyset & \text{[I-TRUE]}\mathcal{M}_{\mathcal{I}}(\texttt{true}) = \text{State} \\
\text{[I-LOOKUP]}\mathcal{M}_{\mathcal{I}}(v = r) = \{(\sigma, O) \mid \sigma(v) = r\} & \text{[I-REF-EQ]}\mathcal{M}(r_1 = r_2) = \text{State if } \mathsf{eqref}(r_1, r_2),\ \emptyset \text{ else} \\
\text{[I-REF-TP]}\mathcal{M}(r : T) = \text{State if } \mathsf{type}(r) = T,\ \emptyset \text{ else} & \text{[I-REF-STP]}\mathcal{M}(r <: T) = \text{State if } \mathsf{type}(r) <: T,\ \emptyset \text{ else} \\
\text{[I-EMPTY]}\mathcal{M}_{\mathcal{I}}(\mathbf{emp}) = \{(\sigma, \emptyset)\} & \text{[I-SINGLE]}\mathcal{M}_{\mathcal{I}}(r_1.a \mapsto r_2) = \{(\sigma, \{(r_1, a, r_2)\})\} \\
\multicolumn{2}{l}{\text{[I-OBJ]}\mathcal{M}_{\mathcal{I}}(\mathsf{obj}(r)) = \{(\sigma, O) \mid \mathsf{dom}\, O = \{r\} \wedge \mathsf{dom}\,(O(r)) = \mathsf{dom}\,(\mathsf{fields}(\mathsf{type}(r)))\}} \\
\text{[I-APP]}\mathcal{M}_{\mathcal{I}}(p(\overline{r})) = \mathcal{I}(p)(\overline{r}) & \text{[I-NEG]}\mathcal{M}_{\mathcal{I}}(\neg\psi) = \text{State} \setminus \mathcal{M}_{\mathcal{I}}(\psi) \\
\multicolumn{2}{l}{\text{[I-OR]}\mathcal{M}_{\mathcal{I}}(\psi_1 \vee \psi_2) = \mathcal{M}_{\mathcal{I}}(\psi_1) \cup \mathcal{M}_{\mathcal{I}}(\psi_2)} \\
\multicolumn{2}{l}{\text{[I-S-CONJ]}\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) \wedge (\sigma, O_2) \in \mathcal{M}_{\mathcal{I}}(\psi_2)\}} \\
\multicolumn{2}{l}{\text{[I-S-IMPLY]}\mathcal{M}_{\mathcal{I}}(\psi_1 -\!\!* \psi_2) = \{(\sigma, O) \mid \forall O_1 \cdot O_1 \perp O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{I}}(\psi_1) \text{ implies } (\sigma, O_1 * O) \in \mathcal{M}_{\mathcal{I}}(\psi_2)\}} \\
\multicolumn{2}{l}{\text{[I-EX]}\mathcal{M}_{\mathcal{I}}(\exists r \cdot \psi) = \{(\sigma, O) \mid \exists r \in \mathsf{Ref} \cdot (\sigma, O) \in \mathcal{M}_{\mathcal{I}}(\psi)\}}
\end{array}
$$

Figure 2: Semantic function for OOSL with interpretation $\mathcal{I}$

Obviously, $(\mathcal{I}_\Lambda, \leq)$ is a complete lattice. $\perp_\Lambda = \{(p, \perp_{\mathcal{P}_{\mathsf{argc}_\Lambda(p)}}) | p \in \mathsf{dom}\,\Lambda\}$ 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 $\mathcal{M}_{\mathcal{I}} : \Psi \to \mathbb{P}(\text{State})$ for OOSL by rules in **Fig.2**. Note that here $\mathcal{I}$ is an arbitrary interpretation. Clearly, a logic environment 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 $p(\overline{r}) \doteq \psi$ 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 $\mathcal{N}_\Lambda : (\mathcal{S} \to \mathcal{P}) \to (\mathcal{S} \to \mathcal{P})$ where:

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

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 is:

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

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

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

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

$$
\begin{array}{lcl}
\mathcal{N}_\Lambda^0 & = & \perp_\Lambda \\
\mathcal{N}_\Lambda^1 & = & \{(list, \{(\texttt{null}, \mathbf{emp})\})\} \\
\mathcal{N}_\Lambda^2 & = & \{(list, \{(\texttt{null}, \mathbf{emp}), (r, r.a \mapsto \texttt{null})\})\} \\
\mathcal{N}_\Lambda^3 & = & \{(list, \{(\texttt{null}, \mathbf{emp}), (r, r.a \mapsto \texttt{null}), (r, r.a \mapsto r' * r'.a \mapsto \texttt{null})\})\} \\
\ldots & &
\end{array}
$$

We know that the model describes all possible lists of this type.

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

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

Definition 6 (Semantics of Assertions)**.**

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

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 many good properties:

Lemma 1. *New predicate function can be safely added to Λ, without changing the meaning of existing symbols in Λ. Formally, if $\Lambda' = (\Lambda, p(\overline{r}) \doteq \psi)$ is a well-formed logic environment, then:*

$$\mathcal{J}_\Lambda(q) = \mathcal{J}_{\Lambda'}(q) \qquad \textit{for every symbol } q \textit{ defined in } \Lambda.$$

By this lemma, we can easily get:

Lemma 2. *For any given logic environment Λ:* (1) *we can safely add some new definitions to it, without changing the meaning of the symbols already defined in Λ; and* (2) *if symbols $\overline{p}$ defined in Λ are not mentioned in other definitions in Λ, then we can safely remove them, without changing the meaning of the other symbols defined in Λ.*

By the semantics of OOSL, 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 $(\sigma \cup \sigma', O) \models \psi$; and* (2) *if ψ does not contain variables in σ', then $(\sigma - \sigma', O) \models \psi$. Here $\sigma - \sigma'$ denotes $\{(x, r) \in \sigma \mid x \notin \mathsf{dom}\,\sigma'\}$.*

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

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

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

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

Lemma 7.

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

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

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

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

Many properties in Separation Logic also hold in OOSL. For example, rules (axiom schemata) shown in the Section 3 of [27] are all valid here.

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 $O \perp O'$.

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 field names.*

As an 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 Separation Logic, due to the absence of fields.

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 $(\sigma, O_2) \models \psi_2$.*

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

Theorem 1. *For any ψ_1, ψ_2, ψ_3, if ψ_1 and ψ_2 are separated from each other, then*

$$\psi_1 * (\psi_2 \mathrel{-\!\!*} \psi_3) \Leftrightarrow \psi_2 \mathrel{-\!\!*} (\psi_1 * \psi_3).$$

Proof. The proof is as follows:

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

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

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

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

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

$\Leftarrow$: For any σ and O that $(\sigma, O) \models \psi_2 \mathrel{-\!\!*} (\psi_1 * \psi_3)$, for any O_1 that $O_1 \perp O$, if $(\sigma, O_1) \models \psi_2$, then $(\sigma, O_1 * O) \models \psi_1 * \psi_3$. Now we fix this O_1. Because $(\sigma, O_1 * O) \models \psi_1 * \psi_3$, 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, and 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 is arbitrary, and $O = O_2 * O_3$, we conclude that $(\sigma, O) \models \psi_1 * (\psi_2 \mathrel{-\!\!*} \psi_3)$. □

This theorem shows a very useful property when we are sure some parts of the Opool are separated from each other. It is often used in the case, after we reason about an assignment to one or more fields of an object, we need to re-construct the whole object. This can always be done, because the different fields of an object are described by separated assertions.

Separated assertions are very useful in reasoning OO programs. Taking the *Node* class above as an example, it allows us to combine relative fields back to form a whole *Node* object:

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

4. μJava: Syntax and WP Semantics

The basic language we use in the work is a sequential subset of Java, μJava [32]. It contains essential OO features, and takes the reference semantics for variables and fields to reflect the reality of mainstream OO languages. μJava has a clear separation of store and heap operations. It is relatively simple to facilitate theoretical study, and large enough for covering important OO issues, e.g., dynamic binding, object sharing, aliasing, casting, etc.

The syntax of μJava is as follows:

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

here x is a variable, C a class name, a and m field and method names respectively. `Object`, `Null`, `Bool`, `true` and `false` take the same meaning as before, and only `Bool` <: `Bool` holds for the boolean type. We use over-lined form to represent sequences. There are some explanations:

- Expressions have limited forms thus their values depend only on the store. Assignments are limited to a number of special forms, including plain assignment $x := e$, mutation $v.a := e$, and lookup $x := v.a$. We consider *cast* as a part of a special form of assignments. Command $x :=$ `new` $C(\overline{e})$ is also a special form of assignment which creates a new object, builds it with parameters $\overline{e}$ and assigns its reference to variable x. More complex structures can be encoded with some auxiliary variables and/or assignments. We assume all references to fields of current object are decorated with `this`, to make the field references uniformly of the form $v.a$. We can remove this restriction by adding repeated rules.

- The special $C(\overline{T\ z})\{\overline{T\ y}; c\}$ in each class C is the constructor, which has the same name as the class. We assume `return` e only appears as the last statement in non-constructor methods, and assume an internal-variable res for recording the return value in semantic definitions. We require that local variables and res initialized to special nil values (represented as nil) according to their types, i.e., rfalse for `Bool` and rnull for class types.

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

In [7] a static environment is defined, then only well-typed expressions and commands are considered in the formal definitions to simplify the presentation. We follow the idea and define a static environment $\Gamma_G = (\Delta_G, \Theta_G)$ for program G. Typing environment Δ_G (abbr. Δ) records static structural information in G. We often omit the context Δ when it is clear. We use $\mathsf{super}(C_1, C_2)$ to mean that C_2 is the immediate superclass of C_1, thus $T_1 <: T_2$ is the transitive closure of super. On the other hand, we record every method for each class in method lookup environment Θ. We will use notation $\Theta, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\}$ to denote that $m(\overline{z})\{\mathsf{var}\ \overline{y}; c\}$ is a method in class C with parameters $\overline{z}$, local variables $\overline{y}$ and body code c.

In [32], we give rules for constructing Δ and Θ and typing. Because the rules are routine, we omit the details here. Based on the static environment, we can check the type for expressions,

the well-typedness of commands and methods. Because of this, we will consider only the well-formed commands and methods in the rest of the paper. We will use $\Gamma, C, m \vdash e : T$ to state that expression e is of the type T in the method m of class C under environment Γ; and similarly, use $\Gamma, C, m \vdash c : \mathbf{com}$ to state that command c is well-typed.

In a class, method can be of the three kinds: is defined in the class in the first time, is an overriding method, or is an inheriting method. We define some notations to distinguish them:

- $\Gamma \vdash \mathsf{intro}(m, C)$ states that method m is introduced (firstly defined) by class C, says all of C's super classes do not contains m.
- $\Gamma \vdash \mathsf{ovr}(m, C)$ states C override the definition of method m.
- We define $\mathsf{def}(m, C) \triangleq \mathsf{intro}(m, C) \vee \mathsf{ovr}(m, C)$, and $\mathsf{ndef}(m, C) \triangleq \neg\mathsf{def}(m, C)$.
- $\Gamma \vdash \mathsf{inh}(C, m, B)$ states class C inherits method m from its super class, and the definition is provided by class B. That is, for any class T, if $C <: T$, $T <: B$ and $T \neq B$, T does not provide new definition for m. We use $\Gamma \vdash \mathsf{inh}(C, m, -)$ to indicate that C inherits m from some class.

4.1. A WP Semantics for μJava

In this section, we define a *weakest precondition semantics* (WP semantics) for μJava and investigate its properties. As usual, the WP semantics of a command c will be defined as a predicate transformer, which maps any given predicate ψ to the weakest precondition of c with respect to ψ. We define the semantics only for well-typed commands, that is, for any command c in discussion, $\Gamma, C, m \vdash c : \mathbf{com}$ is supposed true. The static necessities ensured by typing will not appear in semantic rules.

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

The WP semantics rules for μJava are given in **Fig. 3**. The semantics of sequential composition, choice, and iteration are routine. Their semantics are given by three rules (SEQ), (COND), and (ITER), where $\mu\phi \cdot f$ denotes the least fix-point of $\lambda\phi \cdot f$. Below we give some explanations to each group of the other rules.

Basic Commands: The semantics of `skip` is the identity transformer. The semantics of the plain assignment $x := e$ is ordinary, due to the restricted expression forms in μJava, and the clear separation of assertion forms for the stores and heaps in OOSL.

If any ψ holds after the mutation $v.a := x$, it is necessary that variable v points to some object that has a field a. The existence of field a is guaranteed by typing. After the assignment, $v.a$ holds the reference which is the value of x. This semantics is defined by rule (MUT). The last part of the rule takes the similar form as in the Separation Logic.

As shown by rule (LKUP), the lookup command $x := v.a$ is similar to the plain assignment. The only pre-requirement for executing this command is that v must point to an object which contains a field a. This existence is also guaranteed by typing.

Type cast is treated by rule (CAST). Here we ask for that the variable v must refer to an object with type N or some subtype of N. Remember that for any type T, `null` $<: T$.

$$[\![\Gamma, C, m \vdash c_1; c_2 : \mathbf{com}]\!] = [\![c_1]\!] \circ [\![c_2]\!] \quad \text{(SEQ)}$$

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

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

$$[\![\Gamma, C, m \vdash \texttt{skip} : \mathbf{com}]\!] = \lambda\psi \cdot \psi \quad \text{(SKIP)}$$

$$[\![\Gamma, C, m \vdash x := e : \mathbf{com}]\!] = \lambda\psi \cdot \psi[e/x] \quad \text{(ASN)}$$

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

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

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

$$[\![\Gamma, C, m \vdash \texttt{return}\ e : \mathbf{com}]\!] = \lambda\psi \cdot \psi[e/\mathsf{res}] \quad \text{(RET)}$$

$$\frac{\Theta, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\}, \quad [\![\Gamma, C, m \vdash c : \mathbf{com}]\!] = f}{[\![\Gamma, C \vdash m : \mathbf{method}]\!] = \lambda\ \texttt{this}, \overline{z} \cdot \lambda\psi \cdot f(\psi)[\overline{\mathsf{nil}}/\overline{y}]} \quad \text{(MTHD)}$$

$$\frac{\begin{array}{c}\Gamma, C, m_0 \vdash v : T, \quad S_1, ..., S_k \text{ are all subtypes of } T, \\ [\![\Gamma, S_i \vdash m : \mathbf{method}]\!] = F_i (i = 1, ..., k)\end{array}}{[\![\Gamma, C, m_0 \vdash x := v.m(\overline{e}) : \mathbf{com}]\!] = \lambda\psi \cdot \exists r \cdot (v = r) \wedge (\bigvee (r : S_i \wedge F_i(r, \overline{e})(\psi[\mathsf{res}/x])))} \quad \text{(INV)}$$

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

Figure 3: WP Semantics for μJava

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

Generally, we consider a method as a parameterized command. It can become a command when a this reference and a set of arguments are provided. Following this idea, we define the semantics of a method as a parameterized predicate transformer with type $\mathcal{PT}_{n+1} \triangleq \mathsf{Ref}^{n+1} \to \mathcal{T}$, where n is the number of the parameters of the method, and an extra one for the current object of the invocation. For a $F : \mathcal{PT}_{n+1}$, when we apply it to a set of references $r_0, r_1, \ldots, r_n$, which stand for the objects referred by `this` and all the arguments, we obtain a predicate transformer $F(r_0, r_1, \ldots, r_n)$. For convenience, we define an abbreviation form that for any expression e,

$$F(r_0, .., e, .., r_n) \triangleq \lambda\psi \cdot \exists r \cdot (e = r) \wedge F(r_0, .., r, .., r_n)(\psi).$$

In this case, we may allow an expression to be written in the instantiation form. We may also accept more than one expressions in this abbreviation. For example, we can see that the form $F(r, \overline{e})$ appears in the last two rules in **Fig. 3**.

We use the notation $[\![\Gamma, C \vdash m : \mathbf{method}]\!]$, or short $[\![C.m]\!]$, to denote the WP semantics of a method m defined in class C under environment Γ. Here m could be C to denote the constructor of class C. Now we are ready to give some explanation for the following rules.

Method: Rule (MTHD) gives the semantics of method and constructor declarations. Here all local variables are replaced with nil values. This means that, on one hand, all local variables are initialized with nil according to the requirements mentioned in the explanation for μJava; on the other hand, this also makes all the local variables inaccessible from outside of the method. So, if a given ψ contains names in $\overline{y}$, we should rename such local variables to avoid it.

If all methods are non-recursive, we can get their parameterized predicate transformers directly. Otherwise, by the rules, we can obtain a group of equations about these parameterized predicate transformers. Paper [11] tells us there exists a least fix-point solution for such a set of equations, and we define the solution as the WP semantics for these methods respectively. So the WP semantics for methods is well-defined.

Method Invocation: Based on the above definition, semantics for method invocation is given by rule (INV) which takes a similar form as the corresponding one in [7]. Here we collect all methods of the subclasses of T in the program (which are determined statically by the program text), and define the weakest precondition as the disjunction of the predicates produced by these subclasses. Note that $r : S_i$ ensures $r \neq \mathsf{rnull}$. When reasoning on a real invocation, this disjunction will be resolved by the type of current object and disappeared. In building the precondition, we replace x with res in ψ, because the invocation can be viewed as two "actions": the first one is the execution of the body of $v.m(\overline{e})$ which stores the return value in res at the end, and the second copies the value to x.

Clearly, this rule demands that the program been reasoned about is a closed program. In this case, our definition can describe the behavior of a method invocation precisely. The closeness of the program is one crucial condition, because only under this condition, the WP semantics can achieve completeness. We will study properties of this WP semantics in **Section 4.2**, including the sound and complete theorems.

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

$$\mathsf{raw}(r, N) \mathrel{\hat{=}} \begin{cases} \mathsf{obj}(r), & N \text{ has no field} \\ r : N \wedge (r.a_1 \mapsto \mathsf{nil}) * .. * (r.a_k \mapsto \mathsf{nil}), & \\ & \{a_1, .., a_k\} \text{ is the set of all attrs of } N \end{cases}$$

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

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

4.2. Properties of the WP Semantics

Now we show some properties of the WP semantics for μJava defined above. We have the following theorems, with all their proofs in our report [30].

The first theorem says that the WP semantics is well-defined, i.e., it forms a well-defined function on all well-typed commands and methods.

Theorem 2. *Suppose we have built* Γ *for program G. For any c in G with* $\Gamma, C, m \vdash c : \mathbf{com}$*, its semantics* $[\![\Gamma, C, m \vdash c : \mathbf{com}]\!]$ *is a total function on all formulas. Additionally, if* $\Gamma, C \vdash m : \mathbf{method}$*, the semantics* $[\![\Gamma, C \vdash m : \mathbf{method}]\!]$ *is a well-defined parameterized predicate transformer.*

The WP semantics is monotonic, that is, the predicate transformer defined by any well-typed commands is a monotonic function. In fact, the monotonicity is essential to get a least fix-point solution for parameterized predicate transformers.

Theorem 3. *Suppose $f : \mathcal{T}$ is a predicate transformer produced by rules in* **Fig. 3***, and ψ, ψ' are any well-formed predicates. If $\psi \Rightarrow \psi'$, then $f(\psi) \Rightarrow f(\psi')$.*

Theorem 4. *Given command c and assertions ψ_1, ψ_2, if $FV(\psi_2) \cap md(c) = \emptyset$, then*

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

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

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

In fact, this theorem is the Frame Rule [27] in the WP style.

Example 1 (Empty Object Creation)**.** Now we give a small example to show how to do verification with the WP semantics defined above. Suppose the body of the constructor of `Object` is `skip`, then by the WP semantics we have:

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

Then we have the following calculation:

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

This indicates that two newly created empty objects are different. In fact, this result also holds for non-empty objects, but the calculation is complicated. □

More examples can be found in our report [30].

Now we give the soundness and completeness theorems of the WP semantics defined above. Due to the page limit, we leave their proofs in our report [30].

Informally, a WP semantics $[\![\bullet]\!]$ is *sound*, if for any command c and predicate ψ, if c executes from a state satisfying the weakest precondition $\psi' = [\![c]\!]\psi$, when it terminates, the final state will satisfy ψ. A WP semantics is *complete*, if it really gives the weakest precondition, that is, if any command c executes from any state s and terminates on a state satisfying a condition ψ, then $[\![c]\!]\psi = \psi'$ holds on state s.

We take **COM** the space of legal commands, and use $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$ to denote configuration transformation of μJava, that says when command c executes from current state (σ, O), after its execution of the state will be (σ', O'). More details about the state transformation (operational semantics for μJava) can be found in our report [30].

Now we give the definitions for the soundness and completeness of a WP semantics.

Definition 8 (Soundness)**.** A WP predicate transformer generator $[\![\bullet]\!] : \mathbf{COM} \to \mathcal{T}$ is **sound**, if and only if for any assertions $\psi, \psi' \in \Psi$ and command $c \in \mathbf{COM}$ satisfying $[\![\Gamma, C, m \vdash c : \mathbf{com}]\!]\psi = \psi'$, we have: For any pair of states (σ, O) and (σ', O'), if $(\sigma, O) \models \psi'$ and $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, then $(\sigma', O') \models \psi$.

Definition 9 (Completeness)**.** A WP predicate transformer generator $[\![\bullet]\!] : \mathbf{COM} \to \mathcal{T}$ is **complete**, if and only if for any two assertions $\psi, \psi' \in \Psi$ and command $c \in \mathbf{COM}$ satisfying $[\![\Gamma, C \vdash c : \mathbf{com}]\!]\psi = \psi'$, we have: For any pair of states (σ, O) and (σ', O'), if $(\sigma', O') \models \psi$ and $\langle c, (\sigma, O)\rangle \rightsquigarrow^* (\sigma', O')$, then $(\sigma, O) \models \psi'$.

In these definitions, we recognize the WP semantics as a generator which produces for each command in **COM** a predicate transformer. In report [30], we give the detailed proofs for the soundness and completeness of our WP semantics, according to the operational semantics of the language. Thus we can conclude that,

Theorem 5. *The WP semantics for* μJava *given in* **Section 4.1** *is both sound and complete.*

5. Specification, Refinement, and Behavioral Subtyping

Because the WP semantics defined above is both sound and complete, we can use it as a theoretical foundation to study various problems related to the semantics of μJava. In our report [30], we prove a set of Hoare-style rules for reasoning about OO programs using the WP semantics, including the Frame Rule which is important in local reasoning. Now we study the *behavioral subtyping* concept. This concept involves many important issues in OO program verification, including method specification, refinement and object invariants.

5.1. Method Specification and Refinement

The *specification for a method* (or a piece of code) is often given as a pair of assertions, i.e., the pre and post conditions. In the following we will use $\{P\}\text{-}\{Q\}$ to denote a specification with precondition P and postcondition Q.

A method $C.m(\overline{z})$ satisfies the specification $\{P\}\text{-}\{Q\}$, if the body code of $C.m$ executes under a state (pre-state) where P holds, then Q will hold on the state (post-state) when $C.m$ terminates. This can be defined based on the WP semantics:

Definition 10 (Method Specification)**.** Given any method $C.m(\overline{z})$, we say that method $C.m$ satisfies specification $\{P\}\text{-}\{Q\}$, written as $\{P\}\, C.m\, \{Q\}$, iff:

$$\forall r, \overline{r'} \cdot P[r, \overline{r'}/\mathtt{this}, \overline{z}] \Rightarrow [\![C.m]\!](r, \overline{r'})(Q[r, \overline{r'}/\mathtt{this}, \overline{z}]).$$

This definition is straightforward and intuitive. Based on this definition, we can define the *refinement* relationship between specifications.

Definition 11 (Refinement of Specifications)**.** We say a specification $\{P_2\}\text{-}\{Q_2\}$ refines another specification $\{P_1\}\text{-}\{Q_1\}$, written $\{P_1\}\text{-}\{Q_1\} \sqsubseteq \{P_2\}\text{-}\{Q_2\}$, iff for any command c, $\{P_2\}\, c\, \{Q_2\}$ implies $\{P_1\}\, c\, \{Q_1\}$.

This definition implies that if $\{P_2\}\text{-}\{Q_2\}$ refines $\{P_1\}\text{-}\{Q_1\}$, then the former can substitute the latter anywhere. This idea follows the natural refinement order defined in [15].

Although this definition for the refinement concept is simple and clear, it is not easy for us to use in practice, because we could hardly have ways to investigate all commands. Here we provide a sound condition for the refinement judgement.

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

In fact, this theorem combines the consequence rule in Hoare Logic and Frame Rule in Separation Logic. It provides a useful way to check refinement relation in OO programs where the heap and heap extension are taken into account.

5.2. Behavioral Subtyping

Now we give our definition for *Behavioral Subtyping* based on above discussion.

Definition 12 (Behavioral Subtype)**.** Given class C and B, we say C is a behavioral subtype of B, written $C \preceq B$, iff, for every client accessible method $B.m$ we have for any specification $\{P\}\text{-}\{Q\}$, $\{P\}\, B.m\, \{Q\}$ implies $\{P\}\, C.m\, \{Q\}$.

This definition requires that subclass obeys superclass's behavior. Clearly, this definition follows the thought of Liskov substitution principle.

From now on, we will turn our focus to develop the practical verification procedures. We develop first verification conditions for a program with behavioral subtyping requirement. At first, we introduce some notations. For a μJava program G, we suppose a specification environment Π_G containing specifications of all methods of classes in consideration (we will omit subscript G in what follows, because this will not make any confusion). Π is a map from a method/constructor to its specification. We will use $\{P\}\, C.m\, \{Q\} \in \Pi$ (or $\{P\}\, C.C\, \{Q\} \in \Pi$ for constructor) to state that $\{P\}\text{-}\{Q\}$ is the specification for method $C.m$ (or constructor of C).

Definition 13 (Satisfaction of Specification)**.** We say a program G satisfies specification environment Π, written $G \models \Pi$, iff for every $\{P\}\, C.m\, \{Q\} \in \Pi$, $\{P\}\, C.m\, \{Q\}$ holds, here m could be the constructor.

This definition leads the following verification conditions for $G \models \Pi$:

Theorem 7. *Given a program G and a specification environment* Π*, we have* $G \models \Pi$*, if following condition holds: for every method specification* $\{P\}\, C.m\, \{Q\} \in \Pi$*,* $\{P\}\, C.m\, \{Q\}$ *holds.*

6. Basic OO Specification and Verification Framework: VeriJ$_0$

Now we begin our development of a framework for verifying OO programs, in which we use the OOSL as the assertion language for the specification. In this section, we develop the basic part of the framework for verifying the basic procedural structures.

6.1. Syntax

For build a verification framework, we need to add and modify some features of μJava. We call the result language VeriJ$_0$:

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

Here category S denotes the specifications for constructors and methods, where OOSL assertions P and Q represent pre and post conditions of methods respectively. On the other side, both method and constructor declarations are extended with specification structures.

On the logic side, we add some facilities to OOSL, where

$$\text{[S-CON]}\ \frac{\texttt{class}\ C\{..C(\overline{T\ z})\ \texttt{requires}\ P;\ \texttt{ensures}\ Q\ \{\overline{T\ y};\ c\}..\}}{\{P\}\,C(\overline{z})\,\{Q\}\in\Pi}$$

$$\text{[S-DEF]}\ \frac{\texttt{class}\ C\{..T\ m(\overline{T\ z})\texttt{requires}\ P;\ \texttt{ensures}\ Q\{\overline{T\ y};\ c\}..\}}{\{P\}\,C.m(\overline{z})\,\{Q\}\in\Pi}$$

$$\text{[S-SINH]}\ \frac{\texttt{class}\ C:B\{..T\ m(\overline{T\ z})\{\overline{T\ y};\ c\}..\}\quad \{P\}\,B.m(\overline{z})\,\{Q\}\in\Pi}{\{P\}\,C.m(\overline{z})\,\{Q\}\in\Pi}\qquad \text{[S-MINH]}\ \frac{\Gamma\vdash \mathsf{ndef}(m,C)\quad \mathsf{super}(C,B)\quad \{P\}\,B.m(\overline{z})\,\{Q\}\in\Pi}{\{P\}\,C.m(\overline{z})\,\{Q\}\in\Pi}$$

Figure 4: Rules for constructing specification environment Π

- A special variable `this` is included which denotes always current object, as well as a variable res used to hold the return value of current method.
- **old**(e) is an expression denotes the value of e evaluated under precondition.

As a predefined rule, if we do not provide precondition (postcondition) for some method, it inherits one from its immediate super class, or takes "`requires true;`"(or `ensures true`") default when nothing to inherite.

We discussed the static environment for μJava in **Section 4** for typing and method lookup, as well as the specification environment **Section 5.2**. Now we introduce formally the specification environment Π_G into our static environment. Π_G records the specifications for all the constructor and method. Now for program G, $\Gamma_G = (\Delta_G, \Theta_G, \Pi_G)$. We list rules for constructing Π_G in **Fig. 4**. We will always omit subscribe G when its is clear. In the premise of rule [S-MINH], we use $\Gamma \vdash \mathsf{ndef}(m, C)$ to say that class C does not define method m, where $\mathsf{ndef}(m, C)$ is defined in last section. All these rules are simple, which do not deserve more explanations.

6.2. The Verification Framework

Now we show the basic part of our verification framework. We will use $\Gamma, C, m \vdash \psi$ to denote that ψ holds under Γ in method $C.m$. Clearly, if this hold, the truth value of ψ must not be affected by commands in $C.m$. We use more often statements of the form $\Gamma, C, m \vdash \{P\}\,c\,\{Q\}$, which denotes that command c in $C.m$ meets specification $\{P\}$–$\{Q\}$, and $\Gamma \vdash \{P\}\,C.m\,\{Q\}$ denotes that method $C.m$ (or constructor C) meets specification $\{P\}\,C.m\,\{Q\} \in \Pi$ under Γ.

If every method of class C is correct, then we say that C is correct. If every class in program G is correct, we say that G is correct. Thus we have the following definition:

Definition 14 (Correctness of a program)**.** Given program G in VeriJ$_0$, and its static environment is $\Gamma = (\Delta, \Theta, \Pi)$, we say G is correct *iff* for every specification $\{P\}\,C.m(\overline{z})\,\{Q\} \in \Pi$, we have $\Gamma \vdash \{P\}\,C.m\,\{Q\}$.

Fig. 5 lists rules for verifying various VeriJ$_0$ commands in method body. Rules **[H-SKIP]**, **[H-ASN]**, **[H-RET]**, **[H-SEQ]**, **[H-COND]**, and **[H-ITER]** are standard as in the Hoare Logic. Rules **[H-MUT]** **[H-LKP]** verify the mutation and lookup commands, which are similar to their correspondents in Separation Logic. **[H-INV]** determines the specification of an invocation by callee's static type, this rule requires that all classes obey the subtyping requirement. At last, rule **[H-NEW]** is for verifying the object creations.

The rules in **Fig. 6** are for verifying constructors and methods. As we discussed before, the methods defined in a program can be divided into three kinds: directly defined, overridden and inherited. **[H-DEF]** is for verifying defined methods, where we just need verify its body.

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

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

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

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

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

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

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

Figure 5: Verification rules for commands

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

$$\text{[H-DEF]}\ \frac{\Gamma \vdash \mathsf{intro}(m, C), \quad \Gamma, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \qquad \Gamma \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge P[\overline{r}/\overline{z}]\}\,c\,\{Q[\overline{r}/\overline{z}]\}}{\Gamma \vdash \{P\}\,C.m(\overline{z})\,\{Q\}}$$

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

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

local variables $\overline{y}$ are not free in P,Q

Figure 6: Verification rules for method body

[H-OVR] is for the overridden methods, here we must verify specification in the subclass refine its correspondent in superclass besides body verification, that is, verifying $\Gamma \vdash \{P_B\}\text{-}\{Q_B\} \sqsubseteq \{P_C\}\text{-}\{Q_C\}$. For an inherited method $C.m$, because both of its body and specification are the same as $B.m$ in C's superclass B, so as long as $B.m$ is correct, we have $C.m$ is correct. **[H-INH]** shows this strategy and indicates that we do not need to reverify inherited methods in VeriJ$_0$.

Fig. 7 lists some additional rules. **[H-THIS]** is simple, while **[H-OLD]** says that if expression e evaluates to r' in pre-state, then $\mathbf{old}(e)$ is r' even the value of e is modified. There is a corresponding rule for constructors, which takes the same form. **[H-CONS]**, **[H-EX]** and **[H-FRAME]** are for consequence, existence and frame, where $FV(R)$ is the set of all program variables (including the internal res) in assertion R, and $MD(c)$ denotes the variables modified by command c. The definitions of FV and MD are routine thus are omitted here.

After building of a verification framework, we need to prove its soundness. Informally, a verification framework is sound, iff $\Gamma \vdash G$ holds for any program G, then every method in G meet its specification. Based on μJava's WP semantics mentioned before, this can be defined as:

Definition 15 (Sound Verification Framework)**.** Given a verification framework $\vdash$, it is sound iff for every program G:

- for every command c, if $\Pi_G \vdash \{P\}\,c\,\{Q\}$, then $P \Rightarrow [\![c]\!]_G Q$;
- for every method $C.m$, if $\Pi_G \vdash \{P\}\,C.m(\overline{z})\,\{Q\}$, then $\forall \overline{r} \cdot P[\overline{r}/\overline{z}] \Rightarrow [\![C.m]\!]_G(\overline{r})(Q[\overline{r}/\overline{z}])$;

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

$$\text{[H-CONS]}\ \ \frac{P \Rightarrow P', \ \ \Pi \vdash \{P'\}\, c\, \{Q'\}, \ \ Q' \Rightarrow Q}{\Gamma \vdash \{P\}\, c\, \{Q\}} \qquad \text{[H-EX]}\ \ \frac{\Gamma \vdash \{P\}\, c\, \{Q\}, \ \ r \text{ is free in } P, Q}{\Gamma \vdash \{\exists r \cdot P\}\, c\, \{\exists r \cdot Q\}} \qquad \text{[H-FRAME]}\ \ \frac{\{P\}\, c\, \{Q\}, \ \ FV(R) \cap MD(c) = \emptyset}{\{P * R\}\, c\, \{Q * R\}}$$

Figure 7: Additional verification rules

```
class Node {
  Node left, right;
  Bool mark, check;
  /* whether left subtree has been visited */
}
void schorr_waite(Node root)
requires root = r_root ∧ utree(r_root);
ensures root = r_root ∧ mtree(r_root);
{
  Node t, p, q, s;
  t := root; p := null;
  while (p ≠ null ∨ (t ≠ null ∧ ¬t.mark)) {
    if (t = null ∨ 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;
    }
  }
}
```

Figure 8: Source code for SWM algorithm in VeriJ$_0$

- for every constructor C, if $\Pi_G \vdash \{P\}\, C(\overline{z})\, \{Q\}$, then

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

By this definition and μJava's WP semantics, we can prove:

Theorem 8. *The verification framework for* VeriJ$_0$ *defined in this section is sound.* □

6.3. Verification Example: Method Body Verification

Now we use an examples to show how to verify VeriJ$_0$ programs. We take the Schorr-Waite Graph Marking Algorithm (SWM) as an example to show how to do method body verification. SWM is a famous algorithm for stack-less graph marking, and is said to be "the first mountain that any formalism for pointer aliasing should climb" [5]. **Fig. 8** gives an implementation of SWM in VeriJ$_0$. Here class *Node* is the graph node class which has four fields: *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. In the listing, we can see some the specification for the method. That parts will be explained below.

The basic idea of SWM is that, during the course to traverse the graph (or tree), the algorithm temporarily reverses the left/right pointers, to form a stack. By this stack, after the algorithm finishes some part of marking, it can go up-ward and then traverses other part of the graph (or tree). When the algorithm goes up, it will restore the pointer to their original states, thus recover the original graph (or tree) at the end. Until that time, it finishes the marking as well.

To verify SWM, 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,

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

Figure 9: User defined predicates for verifying SWM algorithm

for which we must introduce some mathematical concepts for graphs. Yang [29] presented the first work on verifying SWM with Separation 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 illustrative example possibly been included in this paper, here, we simplify the specification in two aspects: We require the input is a tree, and do not care about the tree structure preservation. So we take a specification as: given any unmarked tree, after the execution of SWM, all nodes in the tree are marked. Though this specification is not complete, it is an interesting example to illustrate the usefulness and power of our framework.

At first, we introduce in **Fig. 9** some user-defined assertions. Predicate node specifies a single tree node; mtree(r) and utree(r) specify that the whole tree from r is marked or unmarked, respectively. These two predicates are used in the specification for *schorr_waite* in **Fig. 8**. On the other hand, sbot and sseg talk about the implicit stack and the segment of nodes reachable through the stack. In details, 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 all nodes in its left subtree are marked and its right subtree is null, otherwise, its left subtree is null and all nodes in its right subtree is unmarked. Predicate 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. 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.

From **Fig. 8**, we know the specification for the SWM program is:

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

Here *SWM* represents the body of the function *schorr_waite*. This is the statement which we need to prove.

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):

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

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 is specified by $I_p(r_p)$ which is the part of nodes reachable from the implicit stack where p refers to the top element.

If p is null then this part is empty. The second part is specified by $I_p(r_t)$ which is a tree denoted by t. The nodes in the tree must be all marked or unmarked.

We can simply prove the following facts:

First, the precondition establishes the loop invariant:

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

And, the postcondition holds when the loop ends:

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

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 in the body. Now we give these proofs one by one, the *pop* case, the *swing* case, and then the *push* case:

***Case** Pop.* For this case, the branch condition is $p \neq \mathtt{null} \wedge (t = \mathtt{null} \vee t.mark) \wedge p.check$. We have following deduction:

$$
\begin{array}{l}
\{(\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) * ((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 \\
\qquad\qquad (\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}) * ((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 \\
\qquad\qquad (\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}) * ((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 \\
\qquad\qquad (\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 := \mathtt{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}) * ((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 \\
\qquad\qquad (\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\}
\end{array}
$$

***Case** Swing.* For the swing case, the proof is
similar. In this case, the branching condition is $p \neq \mathtt{null} \wedge (t = \mathtt{null} \vee t.mark) \wedge \neg p.check$.

Thus we have deduction:

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

Case Push:. Here the condition is $t \neq \texttt{null} \wedge \neg t.mark$. We have

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

Although here we carry the proof for SWM only with a simple specification, we can see that the verification framework of VeriJ$_0$ works well. In addition, we see that in the real verification procedure, some auxiliary predicates need to be introduced, especially when the program involves in some mathematical concepts. Thus, a useful framework must allow users to define their own predicates and use them in the verification.

6.4. Class Verification and Problems

In [25], Parkinson verified some typical examples in OO verification. We investigate these examples in this section. **Fig. 10** lists our sample code, but without specifications.

Cell is the base class here which contains a field x and two methods to manipulate x. ReCell inherits Cell and extends it by introducing an additional y to backup x's value. This value is backuped by an overriding method *set*, and is restored by a new method *undo*. ReCell should

```
class Cell : Object {
  Bool x;
  void set(Bool b) { this.x = b; }
  Bool get() { return this.x; }
}
class ReCell : Cell {
  Bool y;
  void set(Bool b) { this.y = this.x; this.x = b; }
  void undo() { this.x = this.y; }
}
```

```
class DCell : Cell {
  void set(Bool b) { x = ¬b; }
}
class TCell : Cell {
  Bool y;
  void set(Bool b) { this.y = b; }
  Bool get() { return this.y; }
}
```

Figure 10: Source Code for Cell

```
class Cell : Object {
  void set(Bool b)
  requires this.x ↦ -;
  ensures this.x ↦ old(b);
  { this.x = b; }
  Bool get()
  requires this.x ↦ b; ensures res = b;
  { return this.x; }
}
class ReCell : Cell {
  void set(Bool b)
  requires this.x ↦ - ∗ this.y ↦ -;
  ensures this.x ↦ old(b)∗
      this.y ↦ old(this.x);
  { this.y = this.x; this.x = b; }
  void undo()
  requires this.x ↦ - ∗ this.y ↦ b;
  ensures this.x ↦ b ∗ this.y ↦ b;
  { this.x = this.y; }
}
```

```
class DCell : Cell {
  void set(Bool b)
  requires this.x ↦ -; ensures this.x ↦ ¬old(b);
  { x = ¬b; }
}
class TCell : Cell {
  void set(Bool b)
  requires this.y ↦ -; ensures this.y ↦ old(b);
  { this.y = b; }
  Bool get()
  requires this.y ↦ b; ensures res = b;
  { return this.y; }
}
```

Figure 11: The First Specification for Cell Program

be a behavioral subtype of Cell. DCell inherits Cell, but exhibits a different behavior: Cell.*set*(*b*) stores *b* in *x*, but DCell.*set*(*b*) stores ¬*b* instead. So DCell is not a behavioral subtype of Cell. This type of inheritance is called *interface reuse*. TCell maintains behavior of *set* and *get*, but its inner state is very different from Cell. TCell use a new field *y* in implementing *set* and *get*.

Now we consider how to specify Cell and its subclasses, and if we can verify their subtype relationship correctly.

6.4.1. The First Attempt

It is not hard to provide specifications for the methods, as shown in **Fig. 11**. These specifications seems make sense, and we can straightforwardly prove that every method meets its specification. Now we focus on their subtype relationship.

For ReCell, we need to show that ReCell.*set*'s specification refines Cell.*set*'s, that is:

$$\{\texttt{this}.x \mapsto -\}\text{-}\{\texttt{this}.x \mapsto \textbf{old}(b)\} \sqsubseteq \{\texttt{this}.x \mapsto - * \texttt{this}.y \mapsto -\}\text{-}\{\texttt{this}.x \mapsto \textbf{old}(b) * \texttt{this}.y \mapsto \textbf{old}(\texttt{this}.x)\}$$

Unfortunately, this refinement can not be proved, because it does not hold! The reason is that

```
class Cell : Object {
  void set(Bool b)
  requires obj(this); ensures this.x ↪ old(b);
  { this.x = b; }
  Bool get()
  requires obj(this); ensures res = old(this.x);
  { return this.x; }
}
class ReCell : Cell {
  void set(Bool b)
  requires obj(this);
  ensures this.x ↪ old(b) ∧ this.y ↪ old(this.x);
  { this.y = this.x; this.x = b; }
  void undo()
  requires obj(this); ensures this.x ↪ old(this.y);
  { this.x = this.y; }
}
```

```
class DCell : Cell {
  void set(Bool b)
  requires obj(this);
  ensures this.x ↪ ¬old(b);
  { x = ¬b; }
}
class TCell : Cell {
  void set(Bool b)
  requires obj(this);
  ensures this.y ↪ old(b);
  { this.y = b; }
  Bool get()
  requires obj(this);
  ensures res = old(this.y);
  { return this.y; }
}
```

Figure 12: The Second Specifications for Cell

the specification of ReCell.*set* mentions more storage, this does not conform to the definition of refinement **Defintion 11**.

For DCell, we have the following incorrect refinement:

$$\{\texttt{this}.x \mapsto \texttt{-}\}\text{-}\{\texttt{this}.x \mapsto \textbf{old}(b)\} \sqsubseteq \{\texttt{this}.x \mapsto \texttt{-}\}\text{-}\{\texttt{this}.x \mapsto \neg\textbf{old}(b)\}$$

So, we can not judge if class DCell is a behavioral subtype of Cell or not in VeriJ$_0$.

At last, for TCell, we need prove:

$$\begin{array}{rcl} \{\texttt{this}.x \mapsto \texttt{-}\}\text{-}\{\texttt{this}.x \mapsto \textbf{old}(b)\} & \sqsubseteq & \{\texttt{this}.y \mapsto \texttt{-}\}\text{-}\{\texttt{this}.y \mapsto \textbf{old}(b)\}, \quad \text{and} \\ \{\texttt{this}.x \mapsto b\}\text{-}\{\texttt{res} = b\} & \sqsubseteq & \{\texttt{this}.y \mapsto \texttt{-}\}\text{-}\{\texttt{res} = b\} \end{array}$$

Clearly, as for ReCell, these two refinement relationships can not be proved.

We carefully investigate the precondition of Cell.*set*, $\texttt{this}.x \mapsto \texttt{-}$. This assertion requires that this method depends on field $\texttt{this}.x$, and excludes anything else in current Opool. This precondition is too strong because it abandons any memory extension. This example tells us that we must consider some proper extension for our specification.

6.4.2. The Second Attempt

One simple idea is using looser assertions like $\texttt{this}.x \hookrightarrow \texttt{-}$ to replace $\texttt{this}.x \mapsto \texttt{-}$, because $\texttt{this}.x \hookrightarrow \texttt{-} \Leftrightarrow (\texttt{this}.x \mapsto \texttt{-} * \texttt{true})$, $\texttt{this}.x \hookrightarrow \texttt{-}$ holds on any extension of $\texttt{this}.x \mapsto \texttt{-}$. But this does not work either, because although $\texttt{this}.x \hookrightarrow \texttt{-}$ allow memory extension, it can not ensure that necessary cells are in current Opool, for example, the field *y* of ReCell.

In fact, in practice we often need to talk about *the whole object* but not only some particular fields. Take method *set* and *get* as examples, we hope our specification is "they can execute on any object of Cell or its subclass, and *get* always return latest value set by *set*". This recalls our assertion form obj(*r*) in OOSL, which just describes a complete object which *r* refers to. We use it to specify Cell and its subclasses, and list our second attempt in **Fig. 12**.

Consider the new specification for Cell.*set*, {obj(this)}-{this.$x \hookrightarrow$ **old**(b)} says that *set* can execute on any complete object of Cell or its subclasses; and after its execution, the value of x will be **old**(b). Inside Cell, we have this : Cell by rule (H-THIS), then by **Lemma 5** we have:

$$\texttt{this} : \mathsf{Cell} \wedge \mathsf{obj}(\texttt{this}) \Leftrightarrow \texttt{this}.x \mapsto \_$$

Clearly we can prove that Cell.*set* meets its new specification, and so for all other methods.

The power of obj(this) comes from that it will take the suitable meaning according to the object type of this. This precondition can be read as "the method can execute on an object of Cell or its subclass". It has multiple meanings, thus gives a polymorphic specification! This recognition is very important in building a useful verification framework for OO programs.

For the subtype relationship related to ReCell, we need to prove

$$\{\mathsf{obj}(\texttt{this})\}\text{-}\{\texttt{this}.x \hookrightarrow \mathbf{old}(b)\} \sqsubseteq \{\mathsf{obj}(\texttt{this})\}\text{-}\{\texttt{this}.x \mapsto \mathbf{old}(b) * \texttt{this}.y \mapsto \mathbf{old}(\texttt{this}.x)\}$$

Clearly, by **Theorem 6**, this refinement holds, so ReCell is a behavioral subtype of Cell.

For DCell, refinement

$$\{\mathsf{obj}(\texttt{this})\}\text{-}\{\texttt{this}.x \mapsto \mathbf{old}(b)\} \sqsubseteq \{\mathsf{obj}(\texttt{this})\}\text{-}\{\texttt{this}.x \mapsto \neg\mathbf{old}(b)\}$$

can not be proved, so we can not judge if DCell is a behavioral subtype of Cell or not still.

For TCell, we also can not prove the following refinement be because their postconditions have nothing to do with each other.

$$\{\mathsf{obj}(\texttt{this})\}\text{-}\{\texttt{this}.x \mapsto \mathbf{old}(b)\} \sqsubseteq \{\mathsf{obj}(\texttt{this})\}\text{-}\{\texttt{this}.y \mapsto \mathbf{old}(b)\}$$

However, we really want to prove this behavior subtype relationship too.

6.4.3. Discussions

From above specification/verification samples, we have the following recognitions:

1. In specification/verification of OO programs, user-defined predicates are indispensable, especially in dealing with recursive object structures. We cannot image that a framework can offer all possible useful predicates. Although many works in this field use some predicates explicitly, almost none of the work give them syntactic positions in the language, not to say their visibility/usability rules in verifications. We find this is a very fruitful problem to investigate, and will give our solution in next section.
2. A subclass may take an implementation totally different from its superclass, but still provide similar behavior in the view of the users, e.g., class TCell. The naive specification mechanism shown above refuses such departures, thus is not flexible enough in supporting OO verification. Polymorphic specification mechanisms, like obj(this), should be provided to support the verification in the present of inheritance and overriding.
3. As another issue, DCell above should not be Cell's subtype behaviorally. But our framework provide no rule for stating that *class C is not B's behavioral subtype*! In fact, this relation should be determined and announced by users to prevent such verification.

From above work, we learn that VeriJ$_0$ is not expressive and power enough, we must introduce user-defined predicates and polymorphic specification into our language.

$$\frac{\texttt{class}\ C\{..\texttt{def}\ [\texttt{public}]\ p(\overline{a}) : \psi; ..\}}{(p(\overline{a}), [\texttt{public}]\,\psi) \in \Phi(C)}\ [\text{S-PDEF}] \qquad \frac{p \text{ is not defined in} C \quad \texttt{class}\ C\ :\ B\,\{\ldots\} \quad (p(\overline{a}), [\texttt{public}]\,\psi) \in \Phi(B)}{(p(\overline{a}), [\texttt{public}]\,\psi) \in \Phi(C)}\ [\text{S-PINH}]$$

Figure 13: Constructing of Specification Predicates Environment Φ

7. Information Hiding and Abstract Specification: VeriJ$_1$

Now we extend VeriJ$_0$ by adding abstract specifications and specification predicates with scope rules to solve issues 1 and 2 given in last section. We call the new language VeriJ$_1$.

7.1. Extension of Language and Static Environment

We add user defined predicates (in our words, *specification predicate*) in VeriJ$_1$ as follows:

$$\begin{array}{rcl} N & ::= & \texttt{public} \\ K & ::= & \texttt{class}\ C : C\{\overline{[N]\,T\,a};\ \overline{P};\ C(\overline{T\,z})\,[S]\,\{\overline{T\,y}; c\}; \overline{M}\} \end{array} \qquad \begin{array}{rcl} P & ::= & \texttt{def}\ [N]\ p(\texttt{this}, \overline{a}) : \psi \end{array}$$

Here are some explanations:

- Access modifier `public` is introduced to decorate fields and specification predicates. The `public` fields are visible everywhere, while the non-`public` ones are visible only in the class or its subclass(es) (similar to the case of `protected` in Java).
- Declaration $\texttt{def}\ p(\texttt{this}, \overline{a}) : \psi$ introduces a specification predicate in current class, where parameter `this` (written explicitly at first) denotes current object. The signature of a predicate is visible everywhere, but its definition (its body) has different visibility rule. A predicate can be declared as `public` thus its definition can be used everywhere. On the other hand, when a non-public p is defined in class C, its body is visible only in C and subclasses of C. In other place, p, or $C.p$ as a complete name, is atomic.
- As a rule, a subclass inherits all predicates declared in its superclass, and can override them. However, in the overriding, the new predicates should take the same case in `public` or not as their counterparts in the super classes.
- We demand that not any public field can appear in non-public predicates, and only public fields can appear in public predicates.

To support specification predicates, we need to extend the static environment, now Γ for VeriJ$_1$ consists of four parts, $\Gamma = (\Delta, \Theta, \Pi, \Phi)$. We call the new Φ *specification predicates environment*, that records all the user-defined predicates. Formally speaking, Φ is a map from class to a set of specification predicates. $\Phi(C)$ gives all names and bodies of the specification predicates defined in C. **Fig. 13** lists rules for constructing Φ, where **[S-PDEF]** says that if predicate p is defined in C (including overridden), then p's definition is in $\Phi(C)$; and **[S-PINH]** says that C inherits its superclass' predicates when it does not overrides them. As shown, Φ also records if a predicate is `public`. In the following, we will use $\Phi(C.p(\texttt{this}, \overline{a})) = [\texttt{public}]\,\psi$ to denote that $(p(\texttt{this}, \overline{a}), [\texttt{public}]\,\psi) \in \Phi(C)$.

Thanks to specification predicates, VeriJ$_1$ is more expressive and powerful than VeriJ$_0$. We can solve many problems encountered in VeriJ$_0$ before.

$$\text{[H-DPRE]}\ \frac{r : B \qquad C <: B, \quad \Phi(B.p(\texttt{this}, \overline{a})) = \psi}{\Gamma, C, m \vdash p(r, \overline{r'}) \Leftrightarrow \psi[r, \overline{r'}/\texttt{this}, \overline{a}]} \qquad \text{[H-SPRE]}\ \frac{C <: B, \quad \Phi(B.p(\texttt{this}, \overline{a})) = \psi}{\Gamma, C, m \vdash B.p(r, \overline{r'}) \Leftrightarrow \mathsf{fix}(B, \psi)[r, \overline{r'}/\texttt{this}, \overline{a}]}$$

$$\text{[H-PDPRE]}\ \frac{r : D, \quad \Phi(D.p(\texttt{this}, \overline{a'})) = \texttt{public}\ \psi}{\Gamma, C, m \vdash p(r, \overline{r'}) \Leftrightarrow \psi[r, \overline{r'}/\texttt{this}, \overline{a'}]} \qquad \text{[H-PSPRE]}\ \frac{\Phi(D.p(\texttt{this}, \overline{a})) = \texttt{public}\ \psi}{\Gamma, C, m \vdash D.p(r, \overline{r'}) \Leftrightarrow \mathsf{fix}(D, \psi)[r, \overline{r'}/\texttt{this}, \overline{a}]}$$

$$\text{[H-INH1]}\ \frac{\Gamma \vdash \mathsf{inh}(C, m, B) \quad \Gamma \vdash \{P\}\, B.m(\overline{z})\, \{Q\}, \quad \Gamma, C, m \vdash \{P\}\text{-}\{Q\} \sqsubseteq \{\mathsf{fix}(B, P)\}\text{-}\{\mathsf{fix}(B, Q)\}}{\Gamma \vdash \{P\}\, C.m(\overline{z})\, \{Q\}}$$

Figure 14: Additional Rules for VeriJ$_1$

7.2. Verification Framework

We take most of the verification rules for VeriJ$_0$ as the rules for VeriJ$_1$, including the rules for commands listed in **Fig. 5**, **Fig. 7**, and **Fig. 6** except the rule **[H-INH]** for verifying inherited methods. **Fig. 14** lists the new rules for VeriJ$_1$.

[H-DPRE] and **[H-SPRE]** define the scope, or visibility, of (non-public) specification predicates. They say if a predicate is visible in a class, then it can be unfolded there. However, they have some dissimilarities. **[H-DPRE]** says if r is of a class D, then in any subclass of D we can unfold $p(r, \overline{r'})$ to its definition. **[H-SPRE]** tells that $D.p(r, \overline{r'})$ is equivalent to its definition in D. Here $\mathsf{fix}(D, \psi)$ gives the *instantiated* assertion for $D.p(\ldots)$, and the definition of fix is

$$\mathsf{fix}(D, \psi) = \begin{cases} \neg\mathsf{fix}(D, \psi'), & \text{if } \psi \text{ is } \neg\psi' \\ \mathsf{fix}(D, \psi_1) \otimes \mathsf{fix}(D, \psi_2), & \text{if } \psi \text{ is } \psi_1 \otimes \psi_2 \\ \exists r \cdot \mathsf{fix}(D, \psi'), & \text{if } \psi \text{ is } \exists r \cdot \psi' \\ D.p(\texttt{this}, \overline{r}), & \text{if } \psi \text{ is } p(\texttt{this}, \overline{r}) \wedge \\ & D.p(\texttt{this}, \overline{a}) \in \mathsf{dom}\,\Phi \\ \psi, & \text{otherwise.} \end{cases}$$

where $\otimes$ can be $\vee$, $*$, or $-\!\!*$. Intuitively, fix replaces names of predicates defined in D to their complete names, and unfold, so it can fix the meaning of an assertion in a class. In other words, this function provides a static and fixed explanation for ψ, according to a given class D.

Notice in **[H-SPRE]**, when unfolding $D.p(r, \overline{r'})$, we have to use $\mathsf{fix}(D, \psi)$ to fix body of p at first, then do the substitution. In fact, **[H-DPRE]** is for dynamic binding of specification predicate, while **[H-SPRE]** for static binding. These two rules allow us to hide implementation details of a class, even they are used in the definition of specification predicates.

Rules **[H-PDPRE]** and **[H-PSPRE]** are similar to **[H-DPRE]** and **[H-SPRE]**, but deal with the public predicates. Comparing to the corresponding rules, they do not restrict the scope.

7.3. Soundness

We give the soundness result of verification framework for VeriJ$_1$ here, the key point is the semantics of specification predicates:

Definition 16 (Semantics of specification predicates)**.** Suppose $p(\texttt{this}, \overline{a})$ is a specification predicate, and $\Phi(C_1, p(\texttt{this}, \overline{a})) = \psi_1, \ldots, \Phi(C_n, p(\texttt{this}, \overline{a})) = \psi_n$ are all its definitions in Φ, we define:

$$p(r, \overline{r'}) \quad \hat{=} \quad \bigvee(r : T_i \wedge \psi_n[r, \overline{r'}/\texttt{this}, \overline{a}]) \quad i = 1, \ldots, n.$$

From this definition, we can see that the semantics of $p(r, \overline{r'})$ is determined by all its definitions and its first argument r's type. So, specification predicates are "abstract" and "polymorphic". This is similar to the "dynamic biding" in OO world.

By this definition, we can easily prove (H-DPRE) and (H-PDPRE) are sound.

Lemma 11. *(H-DPRE) and (H-PDPRE) are sound.*

Then, by the definition of fix, we have:

Lemma 12. *(H-SPRE) are (H-PSPRE) sound.*

Proof. Induction on structure of ψ. □

At last, we prove the soundness of (H-INH1).

Lemma 13. *(H-INH1) is sound.*

Proof. By premise, $C.m$ is inherited from superclass D, and $\Gamma \vdash \{P\}\, D.m(\overline{z})\, \{Q\}$. By body verification of (H-MTHD) and (H-OVR) we have:

$$\Gamma, D, m \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge P[\overline{r}/\overline{z}]\}\, c\, \{Q[\overline{r}/\overline{z}]\},$$

Then

$$\Gamma, C, m \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \mathsf{fix}(D, P)[\overline{r}/\overline{z}]\}\, c\, \{\mathsf{fix}(D, Q)[\overline{r}/\overline{z}]\},$$

This is

$$\Gamma \vdash \{\mathsf{fix}(D, P)\}\, C.m(\overline{z})\, \{\mathsf{fix}(D, Q)\}.$$

At last, by refinement relationship

$$\Gamma, C, m \vdash \{P\}\text{-}\{Q\} \sqsubseteq \{\mathsf{fix}(D, P)\}\text{-}\{\mathsf{fix}(D, Q)\},$$

So $\Gamma \vdash \{P\}\, C.m(\overline{z})\, \{Q\})$. □

Combing soundness of VeriJ$_0$'s verification rules, we can conclude that the verification framework of VeriJ$_1$ is sound.

7.4. Study Cases

In this section, we show by examples how the modular specification and verification can be carried out in our framework.

7.4.1. Reexamine Cell

At first, we reexamine the Cell example in last section except DCell, which is left to next section. **Fig. 15** list Cell, ReCell and TCell with new specifications in VeriJ$_1$. In Cell we define a specification predicate *cell* for describing its behavior, and this predicate is overridden in the declaration of TCell.

By rules (H-THIS) and (H-DPRE), we have $cell(\texttt{this}, v) \Leftrightarrow \texttt{this}.x \mapsto v$ holds in Cell, and $cell(\texttt{this}, v) \Leftrightarrow \texttt{this}.x \mapsto v * \texttt{this}.y \mapsto \texttt{-}$ in ReCell, while $cell(\texttt{this}, v) \Leftrightarrow \texttt{this}.y \mapsto v$ in TCell. Clearly, predicate *cell* is polymorphic. It is straightforward to complete body verification by the rules. Now we investigate verification conditions about overriding and inheritance.

For ReCell.*set*, we must prove the behavioral subtype condition:

$$\{cell(\texttt{this}, \texttt{-})\}\text{-}\{cell(\texttt{this}, \mathbf{old}(b))\} \;\sqsubseteq\; \{cell(\texttt{this}, r)\}\text{-}\{cell(\texttt{this}, \mathbf{old}(b)) \wedge bak(\texttt{this}, r))\}$$

Clear it holds.

```
class Cell : Object {
  Bool x;
  def cell(this, v) : this.x ↦ v;
  void set(Bool b)
  requires cell(this, -);
  ensures cell(this, old(b));
  { this.x = b; }
  Bool get()
  requires cell(this, b); ensures res = b;
  { return this.x; }
}
class ReCell : Cell {
  Bool y;
  def cell(this, v) : this.x ↦ v * this.y ↦ -;
  def bak(this, v) : this.x ↦ - * this.y ↦ v;
  void set(Bool b)
  requires cell(this, r);
  ensures cell(this, old(b)) ∧ bak(this, r);
  { this.y = this.x; this.x = b; }
  void undo()
  requires bak(this, b); ensures cell(this, b);
  { this.x = this.y; }
}
class TCell : Cell {
  Bool y;
  def cell(this, v) : this.y ↦ v;
  void set(Bool b) { this.y = b; }
  Bool get() { return this.y; }
}
```

Figure 15: Specifications for Cell in VeriJ$_1$

For ReCell.*get*, we should verify

$$\{cell(\texttt{this}, -)\}\text{-}\{\mathsf{res} = b\} \sqsubseteq \{\mathsf{fix}(cell(\texttt{this}, -))\}\text{-}\{\mathsf{fix}(\mathsf{res} = b)\}$$

this is

$$\{\texttt{this}.x \mapsto b * \texttt{this}.y \mapsto -\}\text{-}\{\mathsf{res} = b\} \sqsubseteq \{\texttt{this}.x \mapsto b\}\text{-}\{\mathsf{res} = b\}$$

By **Theorem 6**, this refinement relationship holds.

Combine above deductions, we can conclude that ReCell is a behavioral subtype of Cell.

Then we investigate overridden methods TCell.*get* and TCell.*set*. Because their specifications are same as their correspondence in Cell. We have that TCell is a behavioral subtype of Cell.

From this example, we can see that, thanks to specification predicates, especially their polymorphic features, we can abstract implementation details away in VeriJ$_1$. By these more power framework, we may specify and verify more programs, or do the work more naturally.

7.4.2. Queue

Now we investigate a non-trivial example. **Fig. 16** gives the implementation and specifications for two queue classes. Here *Node* defines nodes holding boolean values; *Queue* defines simple queues, in which field *hd* holds a linked list of *Node* objects with a head node, thus the node denoted by *hd.nxt* holds the first value in the queue. Method *enqueue* inserts a value into the queue. Subclass *EQueue* of *Queue* defines a kind of *faster queue*s. A new field *tl* in *EQueue* object points to the last node of its list, and a new *enqueue* definition overrides the old one in *Queue*. We omit `return` in *enqueue* and write its return type as `void` only for convenience, because *enqueue* does not need to return any value.

For the specification to be possible, we extend the assertion language by adding mathematical concept of sequences with boolean elements, here α, β and γ denote sequences:

$$\alpha, \beta, \gamma ::= [] \mid [b] \mid \alpha :: \alpha$$

where [] is the empty sequence; $[b]$ is a singleton; and :: denotes sequence concatenation.

```
class Node : Object {
  public Bool val; public Node nxt;
  def public node(this, v, n) :
      this.val ↦ v * this.nxt ↦ n;
  Node(Bool b)
  requires emp; ensures node(this, old(b), rnull)
  { this.val = b; this.nxt = null; }
}
class Queue : Object {
  Node hd;
  def queue(this, α): ∃r_h · this.hd ↦ r_h *
        list(this, r_h, rnull, [rfalse] :: α)
  def list(this, r1, r2, α) : (α = [] ∧ r1 = r2 ∧ emp) ∨
    (∃r3, b, β · (α = [b] :: β) ∧
     (node(r1, b, r3) * list(this, r3, r2, β)));
  Queue()
  requires emp; ensures queue(this, [])
  { Node x; x = new Node(false); this.hd = x; }
  void enqueue(Bool b)
  requires queue(this, α);
  ensures queue(this, α :: [old(b)])
  { Node p, q, n; p = this.hd; q = p.nxt;
    while (q!=null){p = q; q = p.nxt; }
    n = new Node(b); p.nxt = n; }
  Bool dequeue()
  requires queue(this, [b] :: α);
  ensures res = b ∧ queue(this, α) * true
  { Bool x; Node h, p; h = this.head; p = h.next;
    x = p.value; p = p.next; h.next = p; return x; }
  Bool empty()
  requires queue(this, α);
  ensures queue(this, α) ∧
    ((α = [] ∧ res = true) ∨
     (α ≠ [] ∧ res = false))
  { Node p; Bool b;
    p = this.head; p = p.next;
    if (p==null) b = true;
    else b = false;
    return b;
  }
}
class EQueue : Queue {
  Node tl;
  def queue(this, α) : ∃r, r', β, b·
    ([rfalse] :: α = β :: [b]) ∧
    (this.hd ↦ r * this.tl ↦ r' *
     list(this, r, r', β) * node(r', b, rnull));
  EQueue()
  requires emp; ensures queue(this, [])
  { Node x; x = new Node(false);
    this.hd = x; this.tl = x;
  }
  void enqueue(Bool b)
  { Node p, n;
    p = this.tl; n = new Node(b);
    p.nxt = n; this.tl = n;
  }
}
```

Figure 16: *Queue* and *EQueue* in VeriJ$_1$

Class *Node* defines a `public` predicate *node*(this, v, n). The definition of that predicate is accessible in any client of *Node*.

In *Queue*, we define a specification predicate *queue*, which gives the implementation detail of *Queue* objects: field *hd* refers to a linked list holding sequence [rfalse] :: α, i.e., a list with a head node recording rfalse, and the rest nodes hold values in α sequentially. We can see how this predicate is used to specify methods of *Queue*.

Here we define also an auxiliary predicate *list*(this, r_1, r_2, α). It is used to assert a single linked list segment between r_1 and r_2 which holds α. Note that although this (a *Queue* object) is not really used in *list*, it makes a link to the class where the predicate defines, thus can be used. We may extend the language with *static* predicates to mimic static methods in OO languages to make the specification more nature.

The specification of *Queue.enqueue* says that, if a *Queue* object q holds values α, after q.*enqueue*(b) it will hold α :: [**old**(b)]. On the other hand, the specification of *Queue.dequeue* says, if q holds [b] :: α, after q.*dequeue*() it will hold α, and the return value is b. The "$*$ true" part in ensures of *dequeue* means that after execution, some objects covered by the precondition are thrown. To *dequeue*, that is the node formerly recording the first value of the queue, but has been taken away now. Specification for *empty* is simple. Please note, no specification here

mentions anything in the implementation, thus they are abstract.

In *EQueue*, which is a subclass of *Queue*, according to the modified implementation, we override predicate *queue* to reflect the structures of this class. And we redefine code of *enqueue* but inherit its specification. By rules, now *queue* in the specification refers to the new definition, although the specification is inherited from *Queue*. Please note that, here *list* is not redefined, thus is inherited. It is also allowed to override auxiliary predicates.

Now, we prove correctness of these classes:

Node.Node meets its specification.

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
\mathit{Node}\ (\texttt{Bool}\ b)\ \{ \\
\quad \{b = r_b \wedge \mathsf{raw}(\texttt{this}, \mathit{Node})\} \\
\quad \texttt{this}.\mathit{val} = b;\ \ \texttt{this}.\mathit{nxt} = \texttt{null}; \\
\quad \{b = r_b \wedge \texttt{this}.\mathit{val} \mapsto r_b * \texttt{this}.\mathit{nxt} \mapsto \mathsf{rnull}\} \\
\} \\
\{\texttt{this}.\mathit{val} \mapsto \mathbf{old}(b) * \texttt{this}.\mathit{nxt} \mapsto \mathsf{rnull}\}
\end{array}
$$

Queue.Queue meets its specification.

$$
\begin{array}{l}
\{\mathbf{emp}\} \\
\mathit{Queue}()\ \{ \\
\quad \mathit{Node}\ x; \\
\quad \{x = \mathit{rnull} \wedge \mathsf{raw}(\texttt{this}, \mathit{Queue})\} \\
\quad x = \texttt{new}\ \ \mathit{Node}(\texttt{false}); \\
\quad \{\exists r_h \cdot x = r_h \wedge \mathsf{raw}(\texttt{this}, \mathit{Queue}) * \mathit{node}(r_h, \mathsf{rfalse}, \mathsf{rnull})\} \\
\quad \texttt{this}.\mathit{hd} = x; \\
\quad \{\exists r_h \cdot x = r_h \wedge \texttt{this}.\mathit{hd} \mapsto r_h * \mathit{list}(r_h, \mathsf{rnull}, [\mathsf{rfalse}])\} \\
\} \\
\{\mathit{queue}(\texttt{this}, [])\}
\end{array}
$$

Queue.empty meets its specification.

$\{queue(\texttt{this}, \alpha)\}$
$\texttt{Bool}\ empty()\ \{$
  $Node\ p; \texttt{Bool}\ b;$
  $\{p = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge queue(\texttt{this}, \alpha)\}$
  $p = \texttt{this}.hd;$
  $\{\exists r_1 \cdot p = r_1 \wedge b = \mathsf{rfalse} \wedge \texttt{this}.hd \mapsto r_1 * list(r_1, \mathsf{rnull}, [\mathsf{rfalse}] :: \alpha)\}$
  $p = p.nxt;$
  $\{\exists r_1, r_2 \cdot p = r_2 \wedge b = \mathsf{rfalse} \wedge \texttt{this}.hd \mapsto r_1 * node(r_1, \mathsf{rfalse}, r_2) * list(r_2, \mathsf{rnull}, \alpha)\}$
  $\texttt{if}\ (p == \texttt{null})$
    $\{\exists r_1 \cdot p = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge \texttt{this}.hd \mapsto r_1 * node(r_1, \mathsf{rfalse}, \mathsf{rnull}) * list(\mathsf{rnull}, \mathsf{rnull}, [])\}$
    $b = \texttt{true};$
    $\{\exists r_1 \cdot p = \mathsf{rnull} \wedge b = \mathsf{rtrue} \wedge \texttt{this}.hd \mapsto r_1 * node(r_1, \mathsf{rfalse}, \mathsf{rnull})\}$
    $\{p = \mathsf{rnull} \wedge b = \mathsf{rtrue} \wedge queue(\texttt{this}, [])\}$
  $\texttt{else}$
    $\{\exists r_1, r_2 \cdot p = r_2 \wedge r_2 \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge$
      $\texttt{this}.hd \mapsto r_1 * node(r_1, \mathsf{rfalse}, r_2) * list(r_2, \mathsf{rnull}, \alpha)\}$
    $b = \texttt{false};$
    $\{\exists r_1, r_2 \cdot p = r_2 \wedge r_2 \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge$
      $\texttt{this}.hd \mapsto r_1 * node(r_1, \mathsf{rfalse}, r_2) * list(r_2, \mathsf{rnull}, \alpha)\}$
    $\{\exists r_2 \cdot p = r_2 \wedge r_2 \neq \mathsf{rnull} \wedge b = \mathsf{rtrue} \wedge queue(\texttt{this}, \alpha)\}$
  $\texttt{return}\ b;$
$\}$
$\{queue(\texttt{this}, \alpha) \wedge ((\alpha = [] \wedge \mathsf{res} = \mathsf{rtrue}) \vee (\alpha \neq [] \wedge \mathsf{res} = \mathsf{rfalse}))\}$

Queue.enqueue meets its specification.

$\{queue(\texttt{this}, \alpha)\}$
$\texttt{void}\ enqueue(\texttt{Bool}\ b)\ \{$ // *enqueue* in class *Queue*
  $Node\ p, q, n;$
  $\{b = r_b \wedge p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge n = \mathsf{rnull} \wedge queue(\texttt{this}, \alpha)\}$
  $p = \texttt{this}.hd;\ q = p.nxt;$
  $\{\exists r_1, r_2 \cdot b = r_b \wedge p = r_1 \wedge q = r_2 \wedge n = \mathsf{rnull} \wedge$
    $\texttt{this}.hd \mapsto r_1 * node(r_1, \mathsf{rfalse}, r_2) * list(r_2, \mathsf{rnull}, \alpha)\}$
  $\texttt{while}\ (q\ \texttt{!=}\ \texttt{null})\ \{$
    $\{\exists r_p, r_q, c, \beta, \gamma \cdot p = r_p \wedge q = r_q \wedge ([\mathsf{rfalse}] :: \alpha = \beta :: [c] :: \gamma) \wedge$
      $list(r_1, r_p, \beta) * node(r_p, c, r_q) * list(r_q, \mathsf{rnull}, \gamma)\}$
    $p = q;\ q = p.nxt;$
  $\}$
  $\{\exists r_1, r_2, r_p, \beta, c \cdot b = r_b \wedge p = r_p \wedge q = \mathsf{rnull} \wedge n = \mathsf{rnull} \wedge$
    $([\mathsf{rfalse}] :: \alpha = \beta :: [c]) \wedge \texttt{this}.hd \mapsto r_1 * list(r_1, r_p, \beta) * node(r_p, c, \mathsf{rnull})\}$
  $n = \texttt{new}\ Node(b);\ p.nxt = n;$
  $\{\exists r_1, r_2, r_p, r_n, \beta, c \cdot b = r_b \wedge p = r_p \wedge q = \mathsf{rnull}\ \wedge n = r_n \wedge ([\mathsf{rfalse}] :: \alpha = \beta :: [c]) \wedge$
    $\texttt{this}.hd \mapsto r_1 * list(r_1, r_p, \beta) * node(r_p, c, r_n)\ * node(r_n, r_b, \mathsf{rnull})\}$
  $\{\exists r_1 \cdot b = r_b \wedge \texttt{this}.hd \mapsto r_1 * list(r_1, \mathsf{rnull}, [\mathsf{rfalse}] :: \alpha :: [r_b])\}$
$\}$
$\{queue(\texttt{this}, \alpha :: [\mathbf{old}(b)])\}$

Queue.dequeue meets its specification.

$\{queue(\texttt{this}, [b] :: \alpha)\}$
$\texttt{Bool}\ dequeue()\ \{$
  $\texttt{Bool}\ x;\ \ Node\ h, p;$
  $\{b = \mathsf{rfalse} \wedge h = \mathsf{rnull} \wedge p = \mathsf{rnull} \wedge queue(\texttt{this}, [b] :: \alpha)\}$
  $h = \texttt{this}.hd;\ \ p = h.nxt;$
  $\{\exists r_h, r_p \cdot x = \mathsf{rfalse} \wedge h = r_h \wedge p = r_p \wedge$
    $\texttt{this}.hd \mapsto r_h * node(r_h, \mathsf{rfalse}, r_p) * list(r_p, \mathsf{rnull}, [b] :: \alpha)\}$
  $x = p.val;$
  $\{\exists r_h, r_p \cdot x = b \wedge h = r_h \wedge p = r_p \wedge$
    $\texttt{this}.hd \mapsto r_h * node(r_h, \mathsf{rfalse}, r_p) * list(r_p, \mathsf{rnull}, [b] :: \alpha)\}$
  $p = p.nxt;$
  $\{\exists r_h, r_p, r_1 \cdot x = b \wedge h = r_h \wedge p = r_p \wedge$
    $\texttt{this}.hd \mapsto r_h * node(r_h, \mathsf{rfalse}, r_1) * node(r_1, b, r_p) * list(r_p, \mathsf{rnull}, \alpha)\}$
  $h.nxt = p;$
  $\{\exists r_h, r_p, r_1 \cdot x = b \wedge h = r_h \wedge p = r_p \wedge$
    $\texttt{this}.hd \mapsto r_h * node(r_h, \mathsf{rfalse}, r_p) * list(r_p, \mathsf{rnull}, \alpha) * node(r_1, b, \mathsf{rnull})\}$
  $\{\exists r_h, r_1 \cdot x = b \wedge h = r_h \wedge \texttt{this}.hd \mapsto r_h *$
    $list(r_h, \mathsf{rnull}, [\mathsf{rfalse}] :: \alpha) * node(r_1, b, \mathsf{rnull})\}$
  $\{\exists r_1 \cdot x = b \wedge queue(\texttt{this}, \alpha) * node(r_1, b, \mathsf{rnull})\}$
  $\texttt{return}\ x;$
$\}$
$\{\mathsf{res} = b \wedge queue(\texttt{this}, \alpha) * \texttt{true}\}$

EQueue.EQueue meets its specification.

$\{\mathbf{emp}\}$
$EQueue()\{\ Node\ x;$
  $\{x = \mathsf{rnull} \wedge \mathsf{raw}(\texttt{this}, EQueue)\}$
  $x = \texttt{new}\ Node(\texttt{false});$
  $\{\exists r_1 \cdot x = r_1 \wedge \mathsf{raw}(\texttt{this}, EQueue) * node(r_1, \mathsf{rfalse}, \mathsf{rnull})\}$
  $\texttt{this}.hd = x;\ \texttt{this}.tl = x;$
  $\{\exists r_1 \cdot x = r_1 \wedge \texttt{this}.hd \mapsto r_1 * \texttt{this}.tl \mapsto r_1 * list(r_1, r_1, []) * node(r_1, \mathsf{rfalse}, \mathsf{rnull})\}$
$\}$
$\{queue(\texttt{this}, [])\}$

EQueue.enqueue meets its specification.

$\{queue(\texttt{this}, \alpha)\}$
$\texttt{void}\ enqueue(\texttt{Bool}\ b)\ \{$ // *enqueue* in class *EQueue*
  $Node\ p, n;$
  $\{b = r_b \wedge p = \mathsf{rnull} \wedge n = \mathsf{rnull} \wedge queue(\texttt{this}, \alpha)\}$
  $p = \texttt{this}.tl;\ n = \texttt{new}\ Node(b);$
  $\{\exists r_h, r_t, r_n, \beta, c \cdot b = r_b \wedge p = r_t \wedge n = r_n \wedge ([\mathsf{rfalse}] :: \alpha = \beta :: [c]) \wedge$
    $node(r_n, r_b, \mathsf{rnull}) * (\texttt{this}.hd \mapsto r_h * \texttt{this}.tl \mapsto r_t\ * list(r_h, r_t, \beta) * node(r_t, c, \mathsf{rnull}))\}$
  $p.nxt = n;\ \texttt{this}.tl = n;$
  $\{\exists r_h, r_t, r_n, \beta, c \cdot b = r_b \wedge p = r_t \wedge n = r_n \wedge ([\mathsf{rfalse}] :: \alpha = \beta :: [c]) \wedge$
    $(\texttt{this}.hd \mapsto r_h * \texttt{this}.tl \mapsto r_n * list(r_h, r_t, \beta)\ * node(r_t, c, r_n) * node(r_n, r_b, \mathsf{rnull}))\}$
  $\{\exists r_h, r_t, r_n \cdot b = r_b \wedge$
    $(\texttt{this}.hd \mapsto r_h * \texttt{this}.tl \mapsto r_n * list(r_h, r_n, [\mathsf{rfalse}] :: \alpha) * node(r_n, r_b, \mathsf{rnull}))\}$
$\}$
$\{queue(\texttt{this}, \alpha :: [\mathbf{old}(b)])\}$

Please pay attention, Rule **[H-OVR]** asks also for verifying $\Gamma, C, m \vdash \{P'\}\text{-}\{Q'\} \sqsubseteq \{P\}\text{-}\{Q\}$. Because P'/Q' and P/Q are the same, nothing needs to do here.

EQueue.dequeue meets its specification. For the inherited method *EQueue.dequeue*, by Rule **[H-INH]**, we need to prove that there exists an R such that:

$$\Gamma, EQueue, dequeue \vdash (P \Rightarrow \mathsf{fix}(Queue, P) * R) \wedge\ (\mathsf{fix}(Queue, Q) * R \Rightarrow Q)$$

where P is $queue(\texttt{this}, [b] :: \alpha)$ and Q is $\mathsf{res} = b \wedge queue(\texttt{this}, \alpha) * \texttt{true}$. By definition of fix, we get

$$\begin{array}{ll} & \Gamma, EQueue, dequeue \vdash (P \Rightarrow \mathsf{fix}(Queue, P) * R) \\ \Leftrightarrow & (queue(\texttt{this}, [b] : \alpha) \Rightarrow Queue.queue(\texttt{this}, [b] :: \alpha) * R) \end{array}$$

and then

$$\begin{array}{ll} & \Gamma, EQueue, dequeue \vdash (\mathsf{fix}(Queue, Q) * R \Rightarrow Q) \\ \Leftrightarrow & (Queue.queue(\texttt{this}, \alpha) * R \Rightarrow queue(\texttt{this}, \alpha)) \end{array}$$

So, the key point is to prove

$$\Gamma, EQueue, dequeue \vdash Queue.queue(r, \alpha) * R \Leftrightarrow queue(r, \alpha)$$

Let $R = \exists r_t \cdot r.tl \mapsto r_t$, we have

$$\begin{array}{lll} & \Gamma, EQueue, dequeue \vdash Queue.queue(r, \alpha) * R & \\ \Leftrightarrow & \exists r_h \cdot r.hd \mapsto r_h * list(\texttt{this}, r_h, \mathsf{rnull}, [\mathsf{rfalse}] :: \alpha) * (\exists r_t \cdot r.tl \mapsto r_t) & \\ \Leftrightarrow & \exists r_h, r_t \cdot r.hd \mapsto r_h * r.tl \mapsto r_t * list(\texttt{this}, r_h, \mathsf{rnull}, [\mathsf{rfalse}] :: \alpha) & \Leftrightarrow\ queue(r, \alpha) \end{array}$$

So, we conclude that *EQueue.dequeue* is correct, because it maintains the behavior subtyping relationship. Here we do not need to touch the code, thus no re-verification is necessary.

EQueue.empty meets its specification. Similar to *EQueue.dequeue*.

From these proofs we can see that we use only the specifications locally, especially the specification predicates (except in the inheritance case shown above, where we have also locality), thus we have information hiding.

```
EQueue trans(Queue q)
requires queue(q, α);
ensures queue(old(q), []) ∗
      queue(res, α) ∗ true;
{
  Bool f, t; EQueue eq;
  eq = new EQueue();
  f = q.empty();
  while (¬f){
    t = q.dequeue();
    eq.enqueue(t);
    f = q.empty();
  }
  return eq;
}
```

```
EQueue trans(Queue q)
requires queue(q, α); ensures queue(old(q), []) ∗ queue(res, α) ∗ true;
{
  {q = r_q ∧ queue(r_q, α)}
  Bool f, t; EQueue eq; eq = new EQueue();
  {∃r · q = r_q ∧ eq = r ∧ queue(r_q, α) ∗ queue(r, [])}
  f = q.empty();
  {(α = [] ∧ f = true) ∨ (α ≠ [] ∧ f = false)}
  while (¬f){
    {∃r, β, γ · q = r_q ∧ eq = r ∧ γ ≠ [] ∧ α = β :: γ∧
       queue(r_q, γ) ∗ queue(r, β) ∗ true}
    t = q.dequeue(); eq.enqueue(t); f = q.empty();
  }
  {∃r · q = r_q ∧ eq = r ∧ queue(r_q, []) ∗ queue(r, α) ∗ true}
  return eq;
  {queue(old(q), []) ∗ queue(res, α) ∗ true}
}
```

Figure 17: A client of *Queue* and *EQueue* and its verification

7.4.3. A Client of Queue and EQueue

Now we define a client which using *Queue* and *EQueue* to show how client code can be specified and verified abstractly without referring to any implementation details. In **Fig. 17** (left part), we define a method *trans* which takes a *Queue* as its parameter, transfers all elements of this queue to a new created *EQueue*, and returns the new *EQueue*. We list the proof of *trans* also in **Fig. 17** (right part). In this example, our verification is carried out only with the interface of method *Queue.enqueue* and *EQueue.queue*. This shows that our framework supports both abstraction and modularity.

8. Explicit Code Reuse: VeriJ$_2$

It is widely recognized that the inheritance feature in OO languages can be used for specialization, overriding, and code reuse (i.e., [25]). For the specialization and overriding, we require that the behavior of a subclass should be compatible to the behavior of its superclass (behavior subtyping). While for code reuse, we only intend to share interface and/or code of the superclasses. Recall the Cell example in **Section 6.3**, we see that DCell is a typical code reuse example. Although we can specify and verify ReCell and TCell properly via specification predicates in VeriJ$_1$, we can not specify classes like DCell according to our intensions.

In this section, we propose an approach to distinguish these different usages, or, different relationships between classes. We call *C inherits D* while *C* should be a behavioral subtype of *D*, meanwhile *C reuses D* while *C* need not be a behavioral subtype of *D*, but only utilizes some facilities of *D*. Our purpose is to add explicit some syntactic notations for declaration of code reuse to VeriJ$_1$, in order for the users to announce that classes like DCell need not be a behavioral subtype of its superclass. This forms our new language VeriJ$_2$.

8.1. Syntax

Comparing to VeriJ$_1$, its extension VeriJ$_2$ introduces a new subclass declaration:

$$R \quad ::= \quad \texttt{class}\ C \lhd C\ \{\overline{[N]\,T\,a};\ \overline{P};\ [I;]\,C(\overline{T\,z})\,[S]\,\{\overline{T\,y};c\};\overline{M}\}$$

$$\text{[T-RSUPER]}\ \frac{\texttt{class}\ C\ \lhd\ B\,\{\ldots\}}{\mathsf{super}(C, \texttt{Object})} \qquad \text{[T-REUSE]}\ \frac{\texttt{class}\ C\ \lhd\ B\,\{\ldots\}}{\mathsf{reuse}(C, B)} \qquad \text{[T-REUSE2]}\ \frac{\texttt{class}\ C\ :\ B\,\{\ldots\}}{\mathsf{reuse}(C, B)}$$

$$\text{[T-RMTHD]}\ \frac{\mathsf{reuse}(C, B), \quad (m, T \to \overline{T}) \in \mathsf{methods}(B)}{(m, T \to \overline{T}) \in \mathsf{methods}(B)} \qquad \text{[T-RFIELD]}\ \frac{\mathsf{reuse}(C, B), \quad (a, T) \in \mathsf{fields}(B)}{(a, T) \in \mathsf{fields}(C)}$$

$$\text{[T-RLOCAL]}\ \frac{\mathsf{reuse}(C, B), \quad (m, x : T) \in \mathsf{fields}(B)}{(m, x : T) \in \mathsf{fields}(C)} \qquad \text{[M-REUSE]}\ \frac{\Gamma \vdash \mathsf{ndef}(m, C), \quad \mathsf{reuse}(C, B), \quad (m, \lambda(\overline{z})\{\overline{y}; c\}) \in \Theta(B)}{(m, \lambda(\overline{z})\{\overline{y}; c\}) \in \Theta(C)}$$

$$\text{[S-RSINH]}\ \frac{\texttt{class}\ C \lhd B\{..T\ m(\overline{T\ z})\{\overline{T\ y}; c\}..\} \qquad \{P\}\, B.m(\overline{z})\, \{Q\} \in \Pi}{\{P\}\, C.m(\overline{z})\, \{Q\} \in \Pi} \qquad \text{[S-RMINH]}\ \frac{\Gamma \vdash \mathsf{ndef}(m, C), \quad \mathsf{reuse}(C, B) \qquad \{P\}\, B.m(\overline{z})\, \{Q\} \in \Pi}{\{P\}\, C.m(\overline{z})\, \{Q\} \in \Pi}$$

$$\text{[P-RINH]}\ \frac{p \text{ is not defined in} C \quad \texttt{class}\ C\ \lhd\ B\,\{\ldots\} \quad (p(\overline{a}), [\texttt{public}]\,\psi) \in \Phi(B)}{(p(\overline{a}), [\texttt{public}]\,\psi) \in \Phi(C)}$$

Figure 18: Extension of static environment Γ

Class C reuse D, denoted by $\texttt{class}\ C \lhd D\ \{\ldots\}$, means that C takes all of D's interface and implementation, including D's specification predicates and method specifications, but C's behavior need not to be compatible D. In this case, we do not need to verify the behavior subtype relationship. In fact, inheritance can be seen as a special case of reuse.

8.2. Static Environment and Verification Framework

For VeriJ$_2$, we need to extend VeriJ$_1$'s static environment to support reuse relation. We add a new relation reuse between classes to type environment Δ, while $\mathsf{reuse}(C, B)$ means that C reuse B:

$$\Delta = \langle \mathsf{cnames}, \mathsf{super}, \mathsf{methods}, \mathsf{fields}, \mathsf{locvars}, \mathsf{reuse} \rangle$$

The **[T-]** rules in **Fig. 18** define the extension of Δ. **[T-RSUPER]** says that when C reuse B, C's superclass is `Object` but not B. It is the key rule for making reuse different from the inheritance. **[T-REUSE]** and **[T-REUSE2]** are for constructing relation reuse, and **[T-REUSE2]** shows that inheritance is also reuse. **[T-RFIELD]**, **[T-RLOCAL]**, and **[T-RMTHD]** are simple, they show that when one class reuse another, the former contains all implementation of the latter. We define $\lhd$ is the reflexive transitive closure of reuse.

Similar to Δ, other components Θ, Π, Φ in the static environment also need to be extended to support reuse, and **Fig. 18** lists all their rules.

It may be surprising that VeriJ$_2$ takes the same verification framework as VeriJ$_1$. None new inference rule needs to be introduced for the reuse. Let's recall the verification framework of VeriJ$_0$ and VeriJ$_1$, we say that methods can be of the three kinds: directly defined, overridden and inherited. This is classified by relation super in Δ. From **Fig. 18**, when C reuse B, we set C's direct superclass to `Object`. In this case, in verifying C, we need do nothing with B.

8.3. Recall DCell *Example*

Thanks to explicit code reuse, now we can specify that DCell reuses Cell, while is not the behavioral subtype of Cell. **Fig. 19** gives the new code and specification for DCell. Comparing to the code in **Section 6.3**, we only change the relationship between DCell and Cell.

By the verification framework defined above, we will verify DCell.*get* and DCell.*set* by rule (H-DEF). Then nothing about the refinement relationship in **Section 6.3** need to be verified.

```
class Cell : Object {
  def cell(this, v) : this.x ↪ v;
  void set(Bool b)
  requires cell(this, -); ensures cell(this, old(b));
  { this.x = b; }
  Bool get()
  requires cell(this, b); ensures res = b;
  { return this.x; }
}
class DCell ◁ Cell {
  void set(Bool b)
  requires cell(this, -);
  ensures cell(this, ¬old(b));
  { x = ¬b; }
}
```

Figure 19: DCell reuse Cell

9. Related Work

To develop a full-armed specification and verification framework for OO programs is a long standing goal in CS research community. The work presented here is an attempt in this direction. In this section, we overview some closely related work and make some comparisons.

A well defined abstract memory model is essential for formal studies. People have proposed various models to capture the complicated structures of the state space of OO programs. Major models can be roughly classified as *object graphs*, *access traces*, and *stack-heaps*.

The *object graph* models treat objects and their connections as graphs. Examples in this direction include the topological model [23], or object diagram [31]. Models of this kind are intuitive and always independent of languages. [14] presents an operational semantics based on a graph model. However, a suitable reasoning framework for graph models still does not exist. The *access traces* model was introduced by [12] (for pointer-programs), where objects were identified by sets of traces to them. This kind of models have advantages in alias analysis [6], but seem too abstract for general purpose. [8] attempted to define a general inference framework for a trace model. The *stack-heap* models extend normal store model with an heap (a map from address to values) to represent objects. These models seem low-level, however, they are relatively easy to used for defining semantics of programs. Some works have been done upon such models, e.g. [25]. However, a full accounting of all important OO features is still missing.

Early work used FOL in reasoning OO programs. However, it is well-known that the mutable object structures are hard to handle in this setting. Middelkoop tried first to use separation logic (SL) in [21], where only the memory model is revised, not the assertion language. In the work, separation conjunction $*$ is defined on the object level, hence objects cannot be split. This limits the power of the logic considerably, especially the power of *frame rule*.

Parkinson defined a revised SL for OO in his thesis [26] etc. Although the start ideas are similar to ours, the framework is very different. In Parkinson's work, states are defined as:

$$\begin{array}{lcl} \text{Heaps} & \hat{=} & (\text{OIDS} \times \text{FieldNames} \rightharpoonup_{\text{fin}} \text{Values}) \times (\text{OIDS} \rightharpoonup_{\text{fin}} \text{Class}) \\ \text{Stacks} & \hat{=} & \text{Vars} \to \text{Values} \qquad \text{Interpretations} \;\hat{=}\; \text{AuxVars} \to \text{Values} \\ \text{States} & \hat{=} & \text{Heaps} \times \text{Stacks} \times \text{Interpretations} \end{array} .$$

The first part of a heap in $h \in$ Heaps stores values of objects' fields, and the second part stores their type information. Here an object is not explicit, but only a set of cells with the same id from OIDS (*object id*s). The extra component "Interpretations" records values of logical variables. Taking Interpretations into states looks not nice, because it has no place in practice, and logical variables are used only in verification but not in execution. Operator $*$ separates only the first

part of heaps in states, thus different empty objects can not be represented. As seen, our state model records only information of program variables and objects. We have a novel definition for the separation of heaps, and this is efficient even the heaps contain empty objects.

In addition, the logic in [26] adopts *intuitionistic semantics*, thus assertions preserve true with heap extension. This makes it impossible to express precise specifications about heaps, even the simplest "the heap is empty". Consequently, no precise property about OO programs can be verified. Our logic takes *classical semantics*, thus is more expressive [13]. The precise assertions are default, and intuitionistic assertions can be written easily (ref. to [27]).

As an important and blooming field, the challenges of specification and verification for OO programs extract wide attention recently. [17] is a comprehensive survey for this field. Because OO technology advertises abstraction, modularity, inheritance, reuse, etc., these issues become also the key focuses in the formal studies.

In one important direction, *abstract fields* (or similarly *model field*, *specification variable*) and *pure methods* are used in works on abstract specifications. [9, 18] proposed *abstract specification* and *modular verification*; [22] presented a modular verification framework via *abstract fields*. Smans [28] use similar techniques in *implicit dynamic frames* to specify and verify frame properties. Similar abstraction techniques are also used in the widely known tools like Spec# [3, 2] and JML [16]. However, these concepts often either make restrictions on subclasses, or require to reverify the inherited methods. On the other hand, in most of these work, many important OO features are largely omitted, or not well-addressed, especially features related to the mutable object structures, and modular verification in the present of inheritance.

Parkinson developed in [25] a verification framework based on SL, where many OO features are considered. In the work, a concept of *abstract predicate families* is used for data abstraction. Each method is specified by a pair of *static/dynamic* specifications. Coincidentally, Chin et al. [10] presented a similar work with dual specifications. For each method, the *static* specification describes detailed behavior of the method for verifying the implementation; and the *dynamic* specification describes its interface, serving for verification of invocations. Based on this, both approaches can avoid re-verification of inherited methods in some extents. However, dual specifications will not only increase the workload, but also raise consistent issues.

Our framework can achieve the same ability but avoiding the dual specifications. It supports abstraction and offers modularity for both specification and verification. We specify the methods only on the abstract level, and offer *specification predicates* to link abstract specifications with implementation details. We propose syntactic rules for visibility, inheritance and overriding of specification predicates and method specifications, similar to what general OO languages do for methods. Our approach offers full encapsulation ability for implementation details, and can also avoid reverification of inherited methods.

The specification predicates are different from abstract predicate families in several aspects: Predicates in an abstraction predicate family do not have syntactic position, nor clear interrelations to the OO language where they target, but only link to some class by the type of their first parameter and a tag. The families do not have clear connection with the class hierarchy of the programs. This means that abstract predicate families are aliens from the programs. Oppositely, we give specification predicates clear syntactic position, and define their properties according to that. By single specification, we get rid of repeated expressions for implementation details, and can express the semantic design decision for a class only in the local defined predicates. This feature makes it possible to support, in specifications of programs, the *single point rule*, i.e., *every important design decision should be expressed in exact one point*. This rule is extremely important in programming practice. In addition, we allow recursive defined predicates, and define rules

for them. This is necessary in support complex class classes in practice. And more important, our framework introduces the polymorphism into specification mechanisms to support modular verification of OO programs. None of the former work make this clear before.

10. Conclusion and Future Work

In this paper, we present a carefully-designed framework for the specification and verification of OO programs. It is based on a state model for OO programs, and a novel definition for the separation of object heaps. For building the framework, we define an OO Separation Logic (OOSL) with some new assertion forms. We give a full treatment on user-defined predicates 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 extend our model OO language μJava with specifications step by step, investigate many important specification and verification techniques. The method body verification in VeriJ$_0$ is the basis for practice. Then we introduce the *specification predicates*, as well as modular specification and verification techniques related to this concept in VeriJ$_1$. By these techniques, people can write polymorphic specifications in VeriJ$_1$ so that specifying and verifying program behavior become more naturally. At last, we introduce a new syntax for inheritance to deal with code reuse which is very common in practice. As far as our knowledge, such topic has not been carefully studied before. To summary, our approach captures many important core OO features, and supports both abstraction and modularity in specification and verification naturally.

As for the future work, first, it would be interesting to study properties of OOSL, provide and prove more inference rules for more effectively reasoning OO programs. Second, we also take interests in the connection between OOSL and the original Separation Logic (SL). We conjecture that every proposition holding in SL, when it does not involve in address arithmetic, will hold in SL. If such conjecture holds, we can borrow many useful inference rules from SL. The third, we want to explore potentials of single specification approaches, and compare them with the dual ones. The fourth, we also think about more concepts in existing work for OO and others, such as JML, ACSL [4], CASL [1], and to extend our framework to support such concepts. Now, we think that concepts like *axiom* in ACSL and *structural specifications* in CASL are very useful in program specification and verification. At last, we are going to integrate more formal features, such as class invariant, frame properties and so on, into our framework. On the other hand, we also think about specification and verification problems related to more features in OO practice, such as interface, design patterns, and application frameworks, etc.

References

[1] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, and Bernd Krieg-Br˙ Casl: the common algebraic specification language.

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

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, LNCS, pages 49–69. Springer, 2005.

[4] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto.

[5] Richard Bornat. Proving pointer programs in hoare logic. In *Proceedings of the 5th International Conference on Mathematics of Program Construction*, MPC '00, pages 102–126, 2000.

[6] Marius Bozga, Radu Losif, and Yassine Lakhnech. On logics of aliasing. *SAS 2004*, 3148:344–360, 2004.
[7] A.L.C. Cavalcanti and D. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Trans. on Software Engineering*, 26(8):713–728, 2000.
[8] Yifeng Chen and J W Sanders. A pointer logic for object diagrams. Technical report, International Institute for Software Technology, The United Nations University, 2007.
[9] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35(6).
[10] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 87–99, 2008.
[11] W. H. Hesselink. Predicate-transformer semantics of general recursion. *Acta Informatica*, 26:309–332, February 1989.
[12] C.A.R. Hoare and Jifeng He. A trace model for pointers and objects. *ECOOP'99, Object Oriented Programming*, 1628/1999:344–360, 1999.
[13] Samin S. Ishtiaq and Peter W. O'Hearn. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM.
[14] 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.
[15] Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 2006.
[16] 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.
[17] G.T. Leavens, K.R.M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.
[18] K. Rustan Leino. *Toward reliable modular programs*. PhD thesis, California Institute of Technology, Pasadena, CA, USA, 1995. UMI Order No. GAX95-26835.
[19] K. Rustan M. Leino. Data groups: specifying the modification of extended state. *SIGPLAN Not.*, 33:144–153, October 1998.
[20] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
[21] 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.
[22] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Springer-Verlag, 2002.
[23] James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke. Towards a model of encapsulation. Technical report, Elvis Software Design Research Group, 2003.
[24] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 247–258, 2005.
[25] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 75–86, 2008.
[26] M.J. Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, 2005.
[27] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
[28] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
[29] Hongseok Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001. (Technical Report UIUCDCS-R-2001-2227).
[30] Liu Yijing, Qiu Zongyan, and Long Quan. A weakest precondition semantics for Java. Technical Report 2010-46, School of Math., Peking University, 2010. Avaliable at http://www.mathinst.pku.edu.cn/index.php?styleid=2, Preprints.
[31] Liang Zhao, Xiaojian Liu, Zhiming Liu, and Zongyan Qiu. Graph transformations for object-oriented refinement. *Formal Aspects in Computing*, 21(1):103–131, 2009.
[32] Qiu Zongyan, Wang Shuling, and Long Quan. Sequential μJava: Formal foundations. Technical Report 2007-35, School of Math., Peking University, 2007. Avaliable at http://www.mathinst.pku.edu.cn/index.php?styleid=2, Preprints.