Specification Predicates: Linking Abstract Specification to Implementation

Yijing Liu, Hong Ali, and Qiu Zongyan

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

Abstract. Specification and verification for object oriented (OO) programs remains a great challenge despite of decades' efforts. To address the problem, we propose a novel specification and verification framework, which supports abstraction and offers modularity via a set of scope and inheritance rules, and a concept called *specification predicate*. The framework covers the most important OO features like encapsulation, information hiding, inheritance and polymorphism, while only one specification per method is necessary. It can successfully deal with inheritance, keep still modularity in verification, and avoid re-verification of the implementation. We show how the framework can be integrated into an OO language, and use examples to illustrate how the specification and verification can be carried out in our framework following the structures of OO programs in an abstract and modular way.

Keywords: Object Orientation, Abstraction, Modularity, Specification, Verification, Separation Logic

1 Introduction

Object Orientation (OO) is a mainstream paradigm in software development practice. Recently, more and more attentions have been paid on the reliability and correctness of software systems (and in general, computer-based systems). In this circumstance, development of powerful and useful frameworks for specifying and verifying OO programs becomes even urgent. Core OO features, saying modularity, encapsulation, inheritance, polymorphism, etc. enhance the scalability of software greatly. But these features bring also great difficulties to program verification.

Generally, encapsulation and modularity imply information hiding and invisibility of implementation details; the polymorphism enables dynamic program behaviors. Both dynamic behavior and information hiding cause difficulties to verification, because verification is a static procedure, and detailed verification always needs knowledge of implementation details. To overcome these challenges, people have made many efforts to develop useful specification and verification techniques.

Inheritance and polymorphism play important roles in OO programming. As the first formal concept to deal with the issues, *behavioral subtyping* [13, 14] made an important step. The concept has been thought a useful guide for good OO programming, as well as for relevant specification and verification. An OO program with behavior subtyping feature gives more support to reason its behavior statically. Leavens [8] pointed

```
class Node : Object {
  public Bool val; 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;
  }
}
```

Fig. 1. Example OO code: $Queue$ and $EQueue$

out that behavioral subtyping is equivalent to modular reasoning for OO. It has become an indispensable part of many OO verification frameworks, e.g. [4, 9].

Abstraction has been thought as another key to modular verification in OO field. Typical concept proposed in this direction include "Model Field/Abstract Field" in [6, 11, 15], "Data Group" in [12], "Abstract Predicate Family" in [16, 17], "Specification Variable" in [19] and "Pure Methods" in [18]. Although named differently, they aim all to provide some degree of abstraction. Data abstraction is a part of OO philosophy. People think about abstraction in OO design and programming. This demands that we have to think about and do specification and verification on some abstract levels.

To address these challenges, in this paper, we propose a framework to support specification and verification of OO programs modularly and abstractly. Our methodology originates from some intuitive ideas: Firstly, the OO features, especially encapsulation, information hiding, inheritance and polymorphism, should have their reflections in the formal OO verification framework. Secondly, the design of specification language and structures, as a high-level description for programs, should extract successful experiences and methodologies in programming languages field, and integrate them into the specification parts of the language and programs.

Now we illustrate our ideas on specification and verification for OO programs, with a simple example in the first.

Fig. 1 gives a typical piece of OO code. Here a $Node$ class represents nodes holding boolean data. A $Queue$ class implements a kind of simple queue, whose 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. Another class $EQueue$ is a subclass of $Queue$ which defines a kind of "faster queue", where a new field tl pointing to the last node of its list. A new definition for $enqueue$ is given to override the definition of $enqueue$ in superclass $Queue$.

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

- In OO practice, developers distinguish the interface from the implementation, to prevent close dependence on implementation details and achieve high degree of modularity. The case should be the same in formal specification and verification. For better modularity in formal work, we need a level of abstract specification, which can be written completely independent of the code. As an example, for method $enqueue$, on the abstract level, we need to say only its effect on abstract sequence of values stored in the queue, but nothing about the linked list.
- Having the abstract-level specification, we need a way to link it with the implementation code within the class. Clearly, the linkage may involve some implementation details. However, as in programming practice, we do not want the details to leak out. Thus we should have locality and visibility rules here. We introduce a concept called *specification predicate* to serve as the link. These predicates are declared within a class to connect the abstract specification with the code. They are atomic outside of the class. For example, for $Queue$, we need a specification predicate to link the mathematical concept of sequences to the linked list of $Node$.
- The two issues above are general to programs with data abstraction. For OO programs, we also have to consider inheritance and overriding. For example, when we have $Queue$ well-specified and verified, and introduce subclass $EQueue$, it is better that we can reuse the existing specification and verification as much as possible. Our idea is to introduce the inheritance and overriding into the specification and verification framework. Here the abstract specification and specification predicates play also importance roles. For example, we may let $enqueue$ in $EQueue$ inherit abstract specification from $Queue$, and define another (local) specification predicate to connect this specification to the new implementation of $enqueue$. We allow also the overriding of abstract specifications, to support more complex cases. In the inheritance case, the behavior subtyping holds implicitly, because the method in subclass share the same specification as in the superclass. For overridden specification, additional proofs for behavior subtyping must be done.

We develop our framework to embody these considerations. The main contributions of this work are as follows:

- We develop a framework which supports abstract level specification for information hiding and encapsulation. To support modular specification and verification, we integrate the specification facilities with many OO features, and define rules for inheritance, overriding, encapsulation, and visibility of specifications.
- We propose a concept called *specification predicate*. It serves as the link to connect abstract specification with implementation details, and plays a key role in modular verification of OO programs with information hiding and dynamic behavior.
- To present our ideas, a small OO language with specification features, VeriJ, is defined. The specification language used in VeriJ is an object oriented separation logic

(OOSL) we developed before. By the language design and the embedded verification framework, we show how the encapsulation, information hiding, inheritance, polymorphism, etc. can enhance the formal verification ability of an OO language. We define a set of Hoare-style rules for generating proof obligations.

– We use some examples to illustrate how specification and verification can be carried for OO programs written in VeriJ modularly, and how our approach can bridge the gaps between a verification logic and implementation details smoothly.

In the rest of the paper, we introduce briefly our assertion language, OOSL at first (more details in **Appendix A**); then define a small OO language VeriJ with integrated specification features in Section 3. Section 4 presents the verification framework embodied in VeriJ. We illustrate our ideas for modular specification and verification by examples in Section 5, then discuss some related work and conclude.

2 The Assertion Language: OOSL

Now we give a short introduction to our assertion language used in the work, the OO Separation Logic (OOSL). Some details of the logic are given in **Appendix A**. A complete treatment can be found in [20, 21].

To represent the states of OO programs, we use an extended classical Stack-Heap storage model based upon three basic sets Name, Type and Ref, where: Name is an infinite set of names. Special names $\mathtt{true}, \mathtt{false}, \mathtt{null} \in \mathsf{Name}$ denote boolean constants and null. We assume $\mathsf{dtype}(v)$ gets the static type (declaration type) of constant or variable v. Type is an infinite set of types. $\mathtt{Object}, \mathtt{Null}, \mathtt{Bool} \in \mathsf{Type}$, where $\mathtt{Object}$ is the supertype of all classes, $\mathtt{Null}$ is the subtype of all classes. Ref is an infinite set of references denoting object identities. It contains three constants: rtrue, rfalse refer two $\mathtt{Bool}$ objects respectively, and rnull refers to nothing.

A state $s = (\sigma, O) \in \mathrm{State}$ consists of a store and an object pool (Opool):

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

For any $\sigma \in \mathrm{Store}$, we assume $\sigma\mathtt{true} = \mathsf{rtrue}$, $\sigma\mathtt{false} = \mathsf{rfalse}$ and $\sigma\mathtt{null} = \mathsf{rnull}$. We will use $r, r_1, \ldots$ to denote references, and $a, a_1, \ldots$ for fields of objects. For the program states, we can define their well-typedness (see **Appendix A**).

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

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

where T is a type, v a variable or constant, r_1, r_2 references, $p(\overline{r})$ a user-defined assertion with arguments $\overline{r}$. We use Ψ to denote the set of assertions. Here are some explanations:

– ρ denotes assertions independent of Opools. References are atomic values here. For any two references r_1, r_2, $r_1 = r_2$ holds iff r_1 and r_2 are identical.

– η denotes assertions involving Opools. Empty and singleton assertions take the similar forms as in Separation Logic. As a cell in Opool is a field-value binding of an object (denoted by a reference), the singleton takes the form $r_1.a \mapsto r_2$. In addition, $\mathsf{obj}(r, T)$ indicates that the Opool contains exact an entire object of type T, which r refers to. We write $\mathsf{obj}(r, -)$ when do not care about the type.
– Connectors $*$ and $-\!*$ are borrowed from Separation Logic. $\psi_1 * \psi_2$ means that current opool can be split into two parts, where ψ_1 and ψ_2 hold on two parts respectively. $\psi_1 -\!* \psi_2$ means that if we add an opool satisfying ψ_1 to current opool, then the new opool will satisfy ψ_2.

A fixed-point semantics for assertions in OOSL is given in **Appendix A**.

We use $\psi[v/x]$ (or $\psi[r/x]$) to denote the assertion built from ψ by substituting x with variable or constant v (or reference r), and $\psi[r_1/r_2]$ the assertion built from ψ by substituting free r_2 with r_1. We treat $r = v$ the same as $v = r$, and define $v.a \mapsto r$ as $\exists r' \cdot (v = r' \wedge r'.a \mapsto r)$. For user-defined predicates p, we use $p(..., v, ...)$ to denote $\exists r \cdot (v = r \wedge p(..., r, ...))$. Here are some abbreviations from FOL or Separation Logic:

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

The core language for OOSL is simple, however, the user-defined predicates enhance notably its expressiveness. In practice, we often need to add some mathematical concepts, such as quantitative relation, sequences, etc., to enhance the expressiveness of OOSL. Such extensions are orthogonal with the core, and easy to conduct.

3 An OO language with specification: VeriJ

Now we define a small OO language VeriJ. It is an extension of μJava [22], a subset of Java with essential OO features related to object sharing, updating, and creation. VeriJ has integrated features for specification and verification:

$$\begin{array}{lll} v & ::= & \mathtt{this} \mid x \\ e & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{null} \mid v \\ b & ::= & \mathtt{true} \mid \mathtt{false} \mid e = e \mid \neg b \mid b \vee b \mid b \wedge b \\ T & ::= & \mathtt{Bool} \mid \mathtt{Object} \mid C \\ S & ::= & \mathtt{require}\ \psi; \mathtt{ensure}\ \psi \\ N & ::= & \mathtt{public} \\ P & ::= & \mathtt{def}\ [N]\, p(\mathtt{this}, \overline{a}) : \psi \\ M & ::= & T\ m(\overline{T\ z})\ [S]\ \{\overline{T\ y};\ c;\} \\ K & ::= & \mathtt{class}\ C : C\{\overline{[N]\, T\ a};\ \overline{P};\ C(\overline{T\ z})\ [S]\ \{\overline{T\ y}; c\}; \overline{M}\} \\ c & ::= & \mathtt{skip} \mid x := e; \mid v.a := e; \mid x := v.a; \mid x := (C)v; \\ & \mid & x := v.m(\overline{e}); \mid x := \mathtt{new}\ C(\overline{e}); \mid \mathtt{return}\ e; \\ & \mid & c\ c \mid \{c\} \mid \mathtt{if}\ b\ c\ \mathtt{else}\ c \mid \mathtt{while}\ b\ c \\ G & ::= & K \mid K\ G \end{array}$$

Here x is a variable, C a class name, a and m field and method names, C a class, T a type, and ψ an assertion.

- Built-in class `Object` is the superclass of all classes. The only primitive type `Bool` is not a supertype or subtype of any other type.
- We have `public` and `protected` modifiers. For the fields, we take `protected` as default. On the other hand, all methods and constructors are `public`.
- Declaration `def` $p(\texttt{this}, \overline{a}) : \psi$ introduces a user-defined predicate, *specification predicate*, into current class, where parameter `this` (written explicitly) denotes current object. A predicate can be `public` thus its definition can be used everywhere. We demand that no public fields appear in non-public predicates, and only public fields in public predicates. [1] A subclass inherits all predicates from its superclass, and can override them. If 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.
- Clauses `require` P; `ensure` Q provide specification for constructors and methods, where P and Q are the pre and post conditions respectively. Specification predicates are used here. In specifications we can use $\mathbf{old}(e)$ to denote the value of expression e in the pre-state, and use pseudo-variable res to denote the return value. We have inheritance and override of method specifications. If an overridden method is not explicitly specified, it takes the specification from the superclass. On the other hand, if a non-overridden method is not explicitly specified, it takes the default specification "`require true`; `ensure true`".
- We assume `return` e appears only as the last statement of a non-constructor method. In method specifications, we use a pseudo-variable res to denote the return value. We require that all local variables are initialized to nil values (represented as nil) according to their types, i.e., rfalse for `Bool`, and rnull for class types.
- $C(\overline{T\,z})\,[S]\,\{\overline{T\,y};\,c\}$ in class C is the constructor. We assume all references to fields of current object in methods are decorated with `this`, to make field references uniformly to the form $v.a$.

For type checking and verifying program G, we build a static environment $\varGamma_G = (\varDelta_G, \varTheta_G, \varPi_G)$, to record relevant information in G. We omit subscript G when it is clear.

$\varDelta$ records typing information. We write $\mathsf{super}(C, B)$ when B is the direct superclass of C, and $C <: B$ for C is a subtype of B. Further, $\varGamma, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\}$ denotes that $m(\overline{z})\{\mathsf{var}\ \overline{y}; c\}$ is a method defined in class C. We use $\varDelta, C, m \vdash e : T$ to denote the judgment that e is of type T in method m of C under $\varDelta$. Judgments for commands and methods are similar. In [22], we give rules for constructing $\varDelta$ and the typing system.

[1] In fact, `public` predicates can be viewed as "macros" in C/C++, they only offer some convenience but no abstraction. Default predicates abstract non-public fields away, they can be viewed as a new `public` "abstract field". If we had more modifiers, we should define much more syntax and visibility rules for specification predicates too. For example, when we had modifiers `public`, `protected` and `private`, then we would allow specification predicates to be `public`,default or `protected`; and only `public` fields can appear in `public` predicates; only `protected` fields can appear in default specification predicates; only `private` fields can appear in `protected` specification predicates. The underlying thinking for such arrangements is that: If we have abstraction levels 1, 2, . . . , n, then we can pack some elements with level i to an abstract element, which is with level $i + 1$.

$$\frac{\texttt{class}\ C\{..\texttt{def}\ [\texttt{public}]\, p(\overline{a}) : \psi; ..\}}{(p(\overline{a}), [\texttt{public}]\, \psi) \in \Theta(C)} \qquad \text{(P-DEF)}$$

$$\frac{p \text{ not defined in } C \quad \mathsf{super}(C, B) \quad (p(\overline{a}), [\texttt{public}]\, \psi) \in \Theta(B)}{(p(\overline{a}), [\texttt{public}]\, \psi) \in \Theta(C)} \qquad \text{(P-INH)}$$

$$\frac{\texttt{class}\ C\{..C(\overline{T\, z})\ \texttt{require}\ P;\ \texttt{ensure}\ Q\ \{\overline{T\, y}; c\}..\}}{\{P\}\, C(\overline{z})\, \{Q\} \in \mathit{\Pi}} \qquad \text{(S-CON)}$$

$$\frac{\texttt{class}\ C\{..T\ m(\overline{T\, z})\texttt{require}\ P;\ \texttt{ensure}\ Q\{\overline{T\, y}; c\}..\}}{\{P\}\, C.m(\overline{z})\, \{Q\} \in \mathit{\Pi}} \qquad \text{(S-DEF)}$$

$$\frac{\texttt{class}\ C : B\{..T\ m(\overline{T\, z})\{\overline{T\, y}; c\}..\} \quad \{P\}\, B.m(\overline{z})\, \{Q\} \in \mathit{\Pi}}{\{P\}\, C.m(\overline{z})\, \{Q\} \in \mathit{\Pi}} \qquad \text{(S-SINH)}$$

$$\frac{m \text{ not defined in } C \quad \mathsf{super}(C, B) \quad \{P\}\, B.m(\overline{z})\, \{Q\} \in \mathit{\Pi}}{\{P\}\, C.m(\overline{z})\, \{Q\} \in \mathit{\Pi}} \qquad \text{(S-MINH)}$$

Fig. 2. Constructing predicate and specification environments

We omit the details here, because it is rather routine. In the following, we will consider only the well-typed programs.

$\Theta(C)$ records the set of specification predicates defined in C. Fig. 2 gives two rules for constructing Θ: (P-DEF) says that if p is defined or overridden in C, its definition is recorded for C; (P-INH) says if C inherits p from its superclass, then p is recorded for C. We record `public` tag if existing. We demand the overriding predicate must have same `public` status as what of the overridden one. We will write $\Theta(C.p(\texttt{this}, \overline{a})) = [\texttt{public}]\, \psi$ if $(p(\texttt{this}, \overline{a}), [\texttt{public}]\, \psi) \in \Theta(C)$.

Thanks to predicate specifications, we can have abstract assertions, whose local meanings depend on specific class, in fact, depend on the local definitions of specification predicates appeared in the assertion.

Specification environment $\mathit{\Pi}$ records all methods specifications. Rules for constructing $\mathit{\Pi}$ are given in Fig. 2 too (the S-rules). We use $\{P\}\, C.m(\overline{z})\, \{Q\} \in \mathit{\Pi}$ (or $\{P\}\, C(\overline{z})\, \{Q\} \in \mathit{\Pi}$, for constructor) to denote that $\{P\}\, C.m(\overline{z})\, \{Q\}$ (or $\{P\}\, C(\overline{z})\, \{Q\}$) is the specification for method m (constructor C). As seen, we have also specification inheritance. For convenience, we sometimes write $\{P\}\, C.m\, \{Q\}$ and $\{P\}\, C\, \{Q\}$ instead of $\{P\}\, C.m(\overline{z})\, \{Q\}$ and $\{P\}\, C(\overline{z})\, \{Q\}$.

From these rules, we can see all of the specification predicates, method definitions and their specifications can be inherited, or overridden in our framework.

4 Verification Framework for VeriJ Programs

Based on the environment $\mathit{\Gamma}$, in this section, we define a verification framework for VeriJ programs. Before going into the details, at first, we introduce some notations.

We will use $\mathit{\Gamma}, C, m \vdash \psi$ to denote that assertion ψ holds in method m of class C under $\mathit{\Gamma}$; use $\mathit{\Gamma}, C, m \vdash \{P\}\, c\, \{Q\}$ to denote that, command c in method m satisfies the specification consisting of precondition P and postcondition Q.

If we do not care C and m, or saying they are arbitrary, we omit them usually. For any method specification $\{P\}\,C.m\,\{Q\} \in \mathit{\Pi}$, we write $\mathit{\Gamma} \vdash \{P\}\,C.m\,\{Q\}$ (or $\mathit{\Gamma} \vdash \{P\}\,C\,\{Q\}$), to denote that the method (or constructor) is correct wrt. its specification under $\mathit{\Gamma}$. We sometimes use {P}–{Q} to denote a specification with pre and post condition P and Q.

As discussed before, behavioral subtyping is very important in modular verification. To introduce behavioral subtyping into our framework, we first define the refinement relationship between specifications.

Definition 1 (Refinement of Specifications). *Given specifications $\{P_1\}$–$\{Q_1\}$ and $\{P_2\}$–$\{Q_2\}$, we say that the latter one refines the former, denoted by $\{P_1\}$–$\{Q_1\} \sqsubseteq \{P_2\}$–$\{Q_2\}$, iff there exists an assertion R such that R does not contain program variables, and $(P_1 \Rightarrow P_2 * R) \wedge (Q_2 * R \Rightarrow Q_1)$.* □

This refinement definition is an extension of the formulism in Liskov [14], where the condition for specification refinement is $P_1 \Rightarrow P_2 \wedge Q_2 \Rightarrow Q_1$. We take the storage extension into account here. In fact, our definition follows the *nature refinement order* defined in Leavens [8].

Now we define *correct program*, which demands us to verify that every method in the program meets its specification.

Definition 2 (Correct Program). *Given a program and its static environment $\mathit{\Gamma} = (\mathit{\Delta}, \mathit{\Theta}, \mathit{\Pi})$, we say the program is* correct, *iff for each specification $\{P\}\,C.m(\overline{z})\,\{Q\} \in \mathit{\Pi}$, we have that $\mathit{\Gamma} \vdash \{P\}\,C.m(\overline{z})\,\{Q\}$ holds.* □

In **Fig. 3**, we list the inference rules for basic commands and composition structures, as well as three additional rules.

Rules for `skip` and assignment are simple. Rules (H-MUT) and (H-LKUP) are for mutation and heap lookup. They have the similar basic forms as their counterparts in Separation Logic, except the singleton assertion form in them is different. Rule (H-CAST) is special in OO world for the type casting. Here the precondition proposes a type requirement. Rule (H-RET) is for the return statement, and it states that the return value will be assigned to res. The next three are the rules for composition, which take the same form as the rules in Hoare logic.

(H-CONS) and (H-EX) are rules for consequence and existence. The last one is the frame rule (H-FRAME), which takes directly from Separation Logic. This rule shows that we have also the local reasoning, as in Separation Logic. In the rule, $FV(R)$ denotes the set of all program variables (including internal variable res) in assertion R, and $md(c)$ denotes the variables modified by command c.

In addition to rules in **Fig 3**, the rest and more interesting rules in our verification framework are given in **Fig. 4**, that deal with various higher-level language features.

Rule (H-THIS) is simple. Rule (H-OLD) says that if expression e evaluates to r' in pre-state, then $\mathbf{old}(e)$ records r' even the value of e is modified. There is a corresponding rule for constructors, which takes the same form.

(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

$$\Gamma \vdash \{P\}\,\texttt{skip}\,\{P\} \tag{H-SKIP}$$

$$\Gamma \vdash \{P[e/x]\}\,x := e\,\{P\} \tag{H-ASN}$$

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

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

$$\Gamma \vdash \{v = r \wedge r <: N\}\,x := (N)v\,\{x = r\} \tag{H-CAST}$$

$$\Gamma \vdash \{P[e/\mathsf{res}]\}\,\texttt{return}\ e\,\{P\} \tag{H-RET}$$

$$\frac{\Gamma \vdash \{P\}\,c_1\,\{Q\} \qquad \Pi \vdash \{Q\}\,c_2\,\{R\}}{\Gamma \vdash \{P\}\,c_1; c_2\,\{R\}} \tag{(H-SEQ)}$$

$$\frac{\Gamma \vdash \{b \wedge I\}\,c\,\{I\}}{\Gamma \vdash \{I\}\,\texttt{while}\,b\,c\,\{\neg b \wedge I\}} \tag{H-ITER}$$

$$\frac{\Gamma \vdash \{b \wedge P\}\,c_1\,\{Q\} \qquad \Gamma \vdash \{\neg b \wedge P\}\,c_2\,\{Q\}}{\Gamma \vdash \{P\}\,\texttt{if}\ b\,c_1\ \texttt{else}\ c_2\,\{Q\}} \tag{H-COND}$$

$$\frac{\Gamma, C, m \vdash P \Rightarrow P' \qquad \Gamma, C, m \vdash \{P'\}\,c\,\{Q'\} \qquad \Gamma, C \vdash Q' \Rightarrow Q}{\Gamma, C, m \vdash \{P\}\,c\,\{Q\}} \tag{H-CONS}$$

$$\frac{\Gamma, C, m \vdash \{P\}\,c\,\{Q\} \qquad r \text{ is free in } P, Q}{\Gamma, C, m \vdash \{\exists r \cdot P\}\,c\,\{\exists r \cdot Q\}} \tag{H-EX}$$

$$\frac{\Gamma, C, m \vdash \{P\}\,c\,\{Q\} \qquad FV(R) \cap md(c) = \emptyset}{\Gamma, C, m \vdash \{P * R\}\,c\,\{Q * R\}} \tag{H-FRAME}$$

Fig. 3. Inference rules for basic commands and compositions

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 $\mathsf{fix}(D, \psi)$ is the *instantiated* assertion for $D.p(\ldots)$ by the definition of fix as

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

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 described in 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.

We have some rules for verifying methods and constructors. Their side conditions ask that local variables $\overline{y}$ do not occur freely in the pre and post conditions. These side conditions can be provided by necessary variable renaming.

Rule (H-MTHD) is for defined methods. It demands to verify that the body command meets the specification. Due to the premise ask only $\{P\}\,C.m(\overline{z})\,\{Q\} \in \Pi$, recursive method definitions are supported.

$$\Gamma, C, m \vdash \mathtt{this} : C \qquad \text{(H-THIS)}$$

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

$$\frac{r : D \qquad C <: D \qquad \Theta(D.p(\mathtt{this}, \overline{a})) = \psi}{\Gamma, C, m \vdash p(r, \overline{r'}) \Leftrightarrow \psi[r, \overline{r'}/\mathtt{this}, \overline{a}]} \qquad \text{(H-DPRE)}$$

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

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

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

$$\frac{\begin{array}{c}\text{No } D \text{ such that } C <: D \text{ and } m \text{ defined in } D \qquad \Gamma, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y};\ c\} \\ \{P\}\,C.m(\overline{z})\,\{Q\} \in \Pi \qquad \Gamma, C, m \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge P[\overline{r}/\overline{z}]\}\,c\,\{Q[\overline{r}/\overline{z}]\}\end{array}}{\Gamma \vdash \{P\}\,C.m(\overline{z})\,\{Q\}} \qquad \text{(H-MTHD)}$$

$$\frac{\begin{array}{c}\Gamma, C, m \twoheadrightarrow \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \qquad \mathsf{super}(C, D) \qquad \Gamma \vdash \{P'\}\,D.m(\overline{z})\,\{Q'\} \\ \{P\}\,C.m(\overline{z})\,\{Q\} \in \Pi \qquad \Gamma, C, m \vdash \{P'\}\text{–}\{Q'\} \sqsubseteq \{P\}\text{–}\{Q\} \\ \Gamma, C, m \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge P[\overline{r}/\overline{z}]\}\,c\,\{Q[\overline{r}/\overline{z}]\}\end{array}}{\Gamma \vdash \{P\}\,C.m(\overline{z})\,\{Q\}} \qquad \text{(H-OVR)}$$

$$\frac{\begin{array}{c}m \text{ is not defined in } C \qquad \mathsf{super}(C, D) \qquad \Gamma \vdash \{P\}\,D.m(\overline{z})\,\{Q\} \\ \Gamma, C, m \vdash \{P\}\text{–}\{Q\} \sqsubseteq \{\mathsf{fix}(D, P)\}\text{–}\{\mathsf{fix}(D, Q)\}\end{array}}{\Gamma \vdash \{P\}\,C.m(\overline{z})\,\{Q\}} \qquad \text{(H-INH)}$$

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

$$\frac{\Gamma, C, m \vdash v : T \quad \{P\}\,T.n(\overline{z})\,\{Q\} \in \Pi}{\begin{array}{l}\Gamma, C, m \vdash \{v = r \wedge \overline{e = r'} \wedge P[r, \overline{r'}/\mathtt{this}, \overline{z}]\} \\ \qquad\qquad x := v.n(\overline{e});\ \{Q[r, \overline{r'}, x/\mathtt{this}, \overline{z}, \mathsf{res}]\}\end{array}} \qquad \text{(H-INV)}$$

$$\frac{\{P\}\,T(\overline{z})\,\{Q\} \in \Pi}{\Gamma, C, m \vdash \{\overline{e = r'} \wedge P[\overline{r'}/\overline{z}]\}\,x := \mathtt{new}\,T(\overline{e});\ \{\exists r \cdot x = r \wedge Q[r, \overline{r'}/\mathtt{this}, \overline{z}]\}} \qquad \text{(H-NEW)}$$

For all rules involving method, we assume $\overline{y}$ are not free in P, Q

Fig. 4. Inference rules for invocation, object creation, etc.

(H-OVR) is for overridden methods. The additional premise $\Gamma \vdash \{P'\}\text{–}\{Q'\} \sqsubseteq \{P\}\text{–}\{Q\}$ ensures *behavioral subtyping*. Notice we should prove the relationship independent of class and method, this indicates that for any client code, the subclass objects can replace superclass's. Rule (H-INH) is for inherited methods. It checks specifically whether the specification of the method in the subclass is compatible with its counterpart in superclass, denoted by $\{P\}\text{-}\{Q\} \sqsubseteq \{\mathsf{fix}(D,P)\}\text{-}\{\mathsf{fix}(D,Q)\}$ in the premise. This rule involves only the method specifications, but not the method body, so the implementations of inherited methods need not to be re-verified.

Rule (H-CONSTR) is for constructors. It is similar to (H-MTHD). Here $\mathsf{raw}(\texttt{this}, C)$ specifies that `this` refers to a newly created raw object of the type C, and command c will have effects on its state. The definition of $\mathsf{raw}(r, N)$ is

$$\mathsf{raw}(r,N) \mathrel{\widehat{=}} \begin{cases} \mathsf{obj}(r,N), & N \textit{ has no field} \\ r : N \wedge (r.a_1 \mapsto \mathsf{nil}) * \cdots * (r.a_k \mapsto \mathsf{nil}), & \\ \quad a_1, \cdots, a_k \textit{ are all fields of } N \end{cases}$$

Finally, we have rules for method invocation and object creation. Because *behavior subtyping* is ensured by rule (H-OVR) and (H-INV), so it is enough that we do the verification by the declare type of variable v.

Clearly, our verification framework is modular, beside the points mentioned above in various places, when we add a new class to the existed program, we just need to verify the new class but need not to re-consider the code in the existed part in the verification procedure.

5 Verification Examples

Now we show by examples how the modular specification and verification can be carried out in our framework.

5.1 Queue and EQueue

Fig. 5 is an extension of the code given in **Fig. 1** with complete code for methods. We add some more methods in class $Queue$ for more illustration, and add specification annotations into the code according to our consideration. We omit `return` in $enqueue$ and write return type as `void` only for convenience, because $enqueue$ does not need return 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.

In $Node$ we define a `public` predicate $node(\texttt{this}, v, n)$, that 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 $[\mathsf{rfalse}] :: \alpha$, 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 $Queue$.

```
class $Node$ : Object {
  public Bool $val$; public $Node$ $nxt$;
  def public $node$(this, $v, n$) :
      this.$val \mapsto v$ $*$ this.$nxt \mapsto n$;
  $Node$(Bool $b$)
  require $\mathbf{emp}$;
  ensure $node$(this, $\mathbf{old}(b)$, rnull)
  { this.$val = b$; this.$nxt$ = null; }
}
class $Queue$ : Object {
  $Node$ $hd$;
  def $queue$(this, $\alpha$) : $\exists r_h \cdot$ this.$hd \mapsto r_h *$
        $list$(this, $r_h$, rnull, [rfalse] $::$ $\alpha$)
  def $list$(this, $r_1, r_2, \alpha$) :
    $(\alpha = [] \wedge r_1 = r_2 \wedge \mathbf{emp}) \vee$
    $(\exists r_3, b, \beta \cdot (\alpha = [b] :: \beta) \wedge$
     $(node(r_1, b, r_3) * list($this$, r_3, r_2, \beta)))$;
  $Queue$()
  require $\mathbf{emp}$; ensure $queue$(this, [])
  { $Node$ $x$; $x$ = new $Node$(false);
    this.$hd = x$; }
  void $enqueue$(Bool $b$)
  require $queue$(this, $\alpha$);
  ensure $queue$(this, $\alpha$ :: [$\mathbf{old}(b)$])
  { $Node$ $p, q, n$;
    $p$ = this.$hd$; $q = p.nxt$;
    while ($q$ != null){$p = q$; $q = p.nxt$; }
    $n$ = new $Node(b)$; $p.nxt = n$;
  }
  Bool $dequeue$()
  require $queue$(this, $[b] :: \alpha$);
  ensure res $= b \wedge$ $queue$(this, $\alpha$) $*$ true
  { Bool $x$; $Node$ $h, p$;
    $h$ = this.$head$; $p = h.next$;
    $x = p.value$; $p = p.next$;
    $h.next = p$; return $x$;
  }
  Bool $empty$()
  require $queue$(this, $\alpha$);
  ensure $queue$(this, $\alpha$)$\wedge$
    (($\alpha = [] \wedge$ res = true)$\vee$
     ($\alpha \neq [] \wedge$ 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, $\alpha$) : $\exists r, r', \beta, b\cdot$
    ([rfalse] $:: \alpha = \beta :: [b]) \wedge$
    (this.$hd \mapsto r *$ this.$tl \mapsto r' *$
     $list$(this, $r, r', \beta$) $*$ $node(r', b$, rnull));
  $EQueue$()
  require $\mathbf{emp}$; ensure $queue$(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.$tl = n$;
  }
}
```

Fig. 5. VeriJ code for $Queue$ and $EQueue$

Here we write an auxiliary predicate $list$(this$, r_1, r_2, \alpha$) for asserting 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 if a $Queue$ object q holds values α, after $q.enqueue(b)$ it will hold $\alpha :: [\mathbf{old}(b)]$; specification of $Queue.dequeue$ says if q holds $[b] :: \alpha$, after $q.dequeue()$ it will hold α, and the return value is b. The $*\,\texttt{true}$ part in `ensure` 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. If we used different implementation, we might also override auxiliary predicates.

Now, we can prove correctness of all the methods in these classes by rules in **Section 4** locally. We list all the proofs below.

***Node.Node* meets its specification:**

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

***Queue.Queue* meets its specification:**

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

```
            $\{queue(\texttt{this}, [])\}$
```

Queue.empty **meets its specification:**

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

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

***Queue.dequeue* meets its specification:**

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

EQueue.EQueue **meets its specification:**

```
$\{\mathbf{emp}\}$
$EQueue()\{$ $Node\ x$;
  $\{x = \mathsf{rnull} \wedge \mathsf{raw}(\texttt{this}, EQueue)\}$
  $x =$ new $Node(\texttt{false})$;
  $\{\exists r_1 \cdot x = r_1 \wedge \mathsf{raw}(\texttt{this}, EQueue) * node(r_1, \mathsf{rfalse}, \mathsf{rnull})\}$
  this.$hd = x$; 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:**

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

Rule (H-OVR) asks also for verifying $\Gamma \vdash \{P'\}\text{-}\{Q'\} \sqsubseteq \{P\}\text{-}\{Q\}$. Because here P' and P, Q' and Q are the same, nothing needs to do here.

***EQueue.dequeue* meets its specification:** Because $EQueue$ inherits $dequeue$ from $Queue$, we should use Rule (H-INH), where only the last premise should be proved. By **Definition 1**, we need to prove that there exists an assertion R that

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

where

$$P = queue(\mathtt{this}, [b] :: \alpha), \qquad Q = \mathsf{res} = b \wedge queue(\mathtt{this}, \alpha) * \mathtt{true}$$

By definition of fix, we have

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

and

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

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

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

So, we have that $EQueue.dequeue$ meets its specification.

***EQueue.empty* meets its specification:** We could prove this following the same way as the proof for $EQueue.dequeue$.

In these proofs, we use only the specifications locally, especially the specification predicates, thus we have information hiding.

5.2 A Client of *Queue* and *EQueue*

Now we show how the client code can be specified and verified on the abstract level, where we do not refer to any implementation details.

In **Fig. 6**, we define a method $trans$, which takes a $Queue$ parameter, transfers all its elements to a new $EQueue$, and returns the new $EQueue$. The proof of this method is as follows:

```
$EQueue\ trans(Queue\ q)$
require $queue(q, \alpha)$;
ensure $queue(\mathbf{old}(q), []) * queue(\mathsf{res}, \alpha) * \mathtt{true}$;
{
  Bool $f, t$;  $EQueue\ eq$;
  $eq =$ new $EQueue()$;  $f = q.empty()$;
  while $(\neg f)\{$ $t = q.dequeue()$;  $eq.enqueue(t)$;  $f = q.empty()$;  }
  return $eq$;
}
```

Fig. 6. A client method using $Queue$ and $EQueue$ and its specification

```
$EQueue\ trans(Queue\ q)\ \{$
  $\{q = r_q \wedge queue(r_q, \alpha)\}$
  Bool $f, t$;  $EQueue\ eq$;
  $eq =$ new  $EQueue()$;
  $\{\exists r \cdot q = r_q \wedge eq = r \wedge queue(r_q, \alpha) * queue(r, [])\}$
  $f = q.empty()$;
  $\{queue(r_q, \alpha) \wedge ((\alpha = [] \wedge f = \mathtt{true}) \vee (\alpha \neq [] \wedge f = \mathtt{false}))\}$
  while $(\neg f)\{$
    $\{\exists r, \beta, \gamma \cdot q = r_q \wedge eq = r \wedge \gamma \neq [] \wedge \alpha = \beta :: \gamma \wedge queue(r_q, \gamma) * queue(r, \beta) * \mathtt{true}\}$
    $\{\exists r, b, \beta, \gamma, \gamma' \cdot q = r_q \wedge eq = r \wedge \gamma \neq [] \wedge \alpha = \beta :: \gamma \wedge \gamma = [b] :: \gamma' \wedge$
      $queue(r_q, \gamma) * queue(r, \beta) * \mathtt{true}\}$
    $t = q.dequeue()$;
    $\{\exists r, b, \beta, \gamma, \gamma' \cdot q = r_q \wedge eq = r \wedge \gamma \neq [] \wedge \alpha = \beta :: \gamma \wedge \gamma = [b] :: \gamma' \wedge$
      $t = b \wedge queue(r_q, \gamma') * queue(r, \beta) * \mathtt{true}\}$
    $eq.enqueue(t)$;
    $\{\exists r, b, \beta, \gamma, \gamma' \cdot q = r_q \wedge eq = r \wedge \gamma \neq [] \wedge \alpha = \beta :: \gamma \wedge \gamma = [b] :: \gamma' \wedge$
      $queue(r_q, \gamma') * queue(r, \beta :: [b]) * \mathtt{true}\}$
    $f = q.empty()$;
    $\{\exists r, b, \beta, \gamma, \gamma' \cdot q = r_q \wedge eq = r \wedge \gamma \neq [] \wedge \alpha = \beta :: \gamma \wedge \gamma = [b] :: \gamma' \wedge$
      $((\gamma' = [] \wedge f = \mathtt{true}) \vee (\gamma' \neq [] \wedge f = \mathtt{false})) \wedge queue(r_q, \gamma') * queue(r, \beta :: [b]) * \mathtt{true}\}$
  }
  $\{\exists r \cdot q = r_q \wedge eq = r \wedge queue(r_q, []) * queue(r, \alpha) * \mathtt{true}\}$
  return $eq$;
```

$\{queue(\mathbf{old}(q), []) * queue(\mathsf{res}, \alpha) * \mathtt{true}\}$
}

In this example, our verification is carried out only with the interface of method $Queue.enqueue$ and $EQueue.queue$. This shows that our framework is both abstract and modular.

5.3 Retrospection

Now we have a look back to the code and specifications in Fig. 5, and focus on the abstract specifications components of the VeriJ program. We can see that:

– For class $CNode$, because we do not intend to encapsulate anything within it, its "abstract specification" is in fact rather concrete. In the specification of its only method, its constructor, we explore all details of the object to the out world. This specification decision is a natural consequence of the design decision for the code, where we take all fields as public.
– On the other hand, for classes $Queue$, and consequently, $EQueue$, we want to have a full encapsulation. In the code, we protect all fields from accessing outside the class. Accordingly, the specification for methods is fully abstract. No information of the implementation is explored out. What we tell for the client is only, you can put values in, get values out in a determined order. Under this specification, we allow any reasonable implementation.
– This tells us, how abstract a specification is, is at the hand of the specifier, just the same case as the code. As program designers, we should, not only decide how to write code modularly, but also how to specify the program modularly, to make the program best for maintainable, modifiable, upgradable, extendable, and verifiable.

Our framework is one important step to provide the designers ability to do OO programming with specification in an intentional way. By the way, the framework is easy to be immigrated to the data abstraction programming world, because where the situation is simpler.

6 Related Work and Conclusion

In recent years, specification and verification of OO programs extracts much attentions, and many techniques related to abstraction and modularity here are developed. [10] is a comprehensive survey for the achievements and challenges in this field.

Abstract fields (or other names like model field, specification variable) and *pure methods* are widely used in writing abstraction specifications. However, these concepts are not expressive enough to specify methods behavior. Leino [6, 11] considers *abstract specification* and *modular verification*, but many OO features are omitted, especially the mutable object structures. Müller [15] develops a modular verification framework via *abstract fields*, but where inherited methods need be reverified. Smans [18, 19] use similar techniques in *implicit dynamic frames* to specify and verify frame properties. These abstraction techniques are also used in tools like Spec# [3, 4] and JML [9].

Parkinson [17] develops a modular verification framework where many OO features are considered. In the work, abstract predicate families are defined for data abstraction. Each method is specified by a pair of "static/dynamic specifications". A dynamic specification describes a method behavior, and a static one describes its implementation. For a method, its static specification must refine its dynamic specification. Further, the dynamic specification of an overridden or inherited method must obey behavioral subtyping principle. In the approach, a problem is that the static specification may break information hiding, because a static specification has to involve implementation details so that verification could be carried out. Chin [7] does a similar work with "static/dynamic specifications", but they do not support enough abstraction. They introduce "partial/full view" of objects for writing extensible dynamic specification. However, the specifications need to involve many implementation details.

In this paper, we develop a framework for specification and verification of OO programs, which supports abstraction and modularity for both specification and verification. We use *specification predicates* defined in class to separate and link the abstract level specification and the implementation details. We propose syntactic rules for visibility, inheritance and overriding of specification predicates and method specifications, similar to the corresponding rules for methods in classes.

Specification predicates are different from abstract predicate families in several aspects: Predicates in an abstraction predicate family do not have structure property, nor clear interrelations, 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 program. This means that abstract predicate families are aliens from the program. Oppositely, we integrate specification predicates with suitable structural characters. In addition, we allow recursive defined predicates, and define rules for them. This is necessary in support complex class such as $Queue$ in Section 5, as well as more complex classes in practice.

Different to "static/dynamic method specifications" approach, we adopt single specification approach, and only allow dynamic specifications in the method interface. By single specification, we can get rid of repeated expressions for implementation details, and then 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*, which is extremely important in programming practice. Our approach offers full encapsulation ability for the implementation details. By a localize function fix, we also avoid to reverify inherited methods. We define a small language VeriJ with these specification features to illustrate how the framework can be integrated into an OO language. In the language, we take the pure reference semantic model, and use OOSL [20] as the assertion language which can precisely describe OO programs' states. We define the inference rules for deriving proof obligations from programs with specifications for verifying VeriJ programs statically. Comparing to existing work, our approach captures core OO features, and supports both abstraction and modularity in specification and verification more naturally.

As the future work, we will focus on some problems. We will think about some more concepts in existing work in OO and others, such as JML, ACSL [5], CASL [1]. And we will extend the specification language of VeriJ to support such concepts. In fact, we think concepts like *axiom* in ACSL and *structural specifications* in CASL are very useful in specifying programs. We will integrate more formal features, such as class invariant, frame properties and so on, into our framework.

References

1. Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brükner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: the common algebraic specification language. *Theoretical Computer Science*, 286(2):153 – 196, 2002.
2. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, New York, NY, USA, 2007.
3. 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.
4. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, LNCS 3362, pages 49–69. Springer, 2005.
5. Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language (pre. V1.2)*, May 2008.
6. Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35(6):583–599, 2005.
7. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with Separation Logic. In *POPL'08*. ACM, 2008.
8. Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical report, 2006.
9. 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.
10. 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.
11. K. Rustan Leino. *Toward reliable modular programs*. PhD thesis, Pasadena, CA, USA, 1995. UMI Order No. GAX95-26835.
12. K. Rustan M. Leino. Data groups: specifying the modification of extended state. *SIGPLAN Not.*, 33:144–153, October 1998.
13. Barbara Liskov. Keynote address - data abstraction and hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM.
14. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
15. P. Müller. *Modular specification and verification of object-oriented programs*. Springer-Verlag, LNCS 2262, 2002.
16. M.J. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL'05*. ACM, 2005.
17. M.J. Parkinson and G.M. Bierman. Separation logic, abstraction and inheritance. In *Principles of Programming Languages (POPL'08)*. ACM, 2008.

18. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings ECOOP 2009*, Genoa. Springer-Verlag, 2009.
19. Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *Proceedings of FASE, volume 4961 of LNCS*. Springer, 2008.
20. Liu Yijing and Qiu Zongyan. A separation logic for OO programs. In *Formal Aspects of Computer Software (FACS 2010)*, 2010.
21. 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/, priprint.
22. 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/, priprint.

A OOSL: Some Details and Semantics

Here we give a short introduction to OOSL (Object Oriented Separation Logic), which is used in this work. A more complete treatment of the logic can be found in [21].

A.1 The Storage Model

To deal with OO programs, we extend the classical Stack-Heap storage model. The model used here is defined upon three basic sets Name, Type and Ref.

- Name: an infinite set of names, used for naming various entities, e.g., constants, variables, fields, etc. Three special names, $\texttt{true}, \texttt{false}, \texttt{null} \in \mathsf{Name}$, denote boolean and null constants. We assume a function $\mathsf{dtype} : \mathsf{Name} \to \mathsf{Type}$, where $\mathsf{dtype}(v)$ is the declaration type of constant, variable, or field v.
- Type: an infinite set of types, including predefined types and user-defined types (or *classes*). Subtype relation is represented by symbol $<:$, where $T_1 <: T_2$ states that T_1 is a subtype of T_2. We assume three predefined types $\texttt{Object}, \texttt{Null}$ and $\texttt{Bool}$, where $\texttt{Object}$ is the super type of all classes, $\texttt{Null}$ is the subtype of all classes, and $\texttt{Bool}$ is the type of boolean objects. Given a type T, we can obtain its fields by function $\mathsf{fields} : \mathsf{Type} \to \mathsf{Name} \to \mathsf{Type}$; and we define $\mathsf{fields}(\texttt{Object}) = \mathsf{fields}(\texttt{Null}) = \mathsf{fields}(\texttt{Bool}) = \emptyset$. We consider only boolean type in this paper. Other predefined types, such as Integer, can be added easily.
- Ref: an infinite set of references representing the identities of objects. Corresponding to the Name constants, Ref contains three basic references $\mathsf{rtrue}, \mathsf{rfalse}$ and rnull, where $\mathsf{rtrue}, \mathsf{rfalse}$ refer two $\texttt{Bool}$ objects respectively, while rnull never refers to any object. We assume two primitive functions on Ref:[2]
  - $\mathsf{eqref} : \mathsf{Ref} \to \mathsf{Ref} \to$ bool, justifies whether two references are same, i.e. for any $r_1, r_2 \in \mathsf{Ref}$, $\mathsf{eqref}(r_1, r_2) =$ true iff r_1 is same to r_2.
  - $\mathsf{type} : \mathsf{Ref} \to \mathsf{Type}$ decides the runtime type of object referred by reference. We define $\mathsf{type}(\mathsf{rtrue}) = \mathsf{type}(\mathsf{rfalse}) = \texttt{Bool}, \mathsf{type}(\mathsf{rnull}) = \texttt{Null}$.

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

The storage model is defined based on above concepts:

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

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

We will use $r, r_1, \ldots$ to denote references, and $a, a_1, \ldots$ to denote 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 corresponding values (also references). When we refer to the domain of O, we are referring to either a subset of Ref associated with a set of objects as discussed above, or a subset of Ref $\times$ Name associated with a set of values (references). We use $\mathsf{dom}\, O$ for the former, and define $\mathsf{dom}_2\, O \mathrel{\widehat{=}} \{(r, a) \mid r \in \mathsf{dom}\, O, a \in \mathsf{dom}\, O(r)\}$ for the later, that is, $\mathsf{dom}_2\, O$ represents all the reference and field pairs of non-empty objects in O.

When considering the program states, we demand some regularities, that is, the well-typedness as follows.

Definition 3 (Well-typed Store). *A store σ is well-typed iff*

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

Definition 4 (Well-typed Opool). *An Opool O is well-typed iff*

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

Here $\mathsf{fld}(r) = \mathsf{dom}\,\mathsf{fields}(r)$ is the field set of the type of r. $\square$

The first condition requires all fields in O to be valid according to types of their objects, and hold values of correct types. The second condition requires that if a non-empty object (according to its type) is in O, then at least one field of the object should be in O. Thus we can identify empty objects in any Opool.

As an example, suppose $\mathsf{dom}\,\mathsf{fields}(C) = \{a_1, a_2, a_3\}$, and $\mathsf{type}(r_1) = \mathtt{Object}$, $\mathsf{type}(r_2) = C$, then $O_1 = \{r_1 \mapsto \emptyset, r_2 \mapsto \{a_1 \mapsto \mathsf{rnull}, a_2 \mapsto \mathsf{rnull}\}\}$ 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 5 (Well-typed State). *A state $s = (\sigma, O)$ is well-typed iff both σ and O are well-typed.* $\square$

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

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

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

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

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

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

That is, if a reference, referring to some object o, is in both $\mathsf{dom}\, O_1$ and $\mathsf{dom}\, O_2$, then both O_1 and O_2 must contain non-empty subsets of o's 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. We will use $O_1 * O_2$ to indicate the union of O_1 and O_2, when $O_1 \perp O_2$.

This OO storage model is simple and intuitive. It gives us both object view and field view for a program state. With this model, we can correctly handle objects and their fields, especially empty objects.

A.2 The Logic

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

$$\begin{aligned} \alpha &::= \mathtt{true} \mid \mathtt{false} \mid r_1 = r_2 \mid r : T \mid r <: T \mid v = r \\ \beta &::= \mathbf{emp} \mid r_1.a \mapsto r_2 \mid \mathsf{obj}(r, T) \\ \psi &::= \alpha \mid \beta \mid p(\overline{r}) \mid \neg\psi \mid \psi \vee \psi \mid \psi * \psi \mid \psi \mathrel{-\!\!*} \psi \mid \exists r \cdot \psi \end{aligned}$$

where T is type names, v is a variable or constant name, r_1, r_2 are references. $p(\overline{r})$ denotes user-defined assertions with real arguments $\overline{r}$. We will use Ψ to denote the set of all predicates.

Basic assertions are of two kinds, namely *primitive assertions* and *user-defined assertions*. Primitive assertions fall into two categories, where

- α denotes assertions that are independent of Opools. References are atomic values in our logic. For any two references r_1, r_2, $r_1 = r_2$ holds iff r_1 and r_2 are identical, i.e., $\mathsf{eqref}(r_1, r_2)$. We treat $r = v$ the same as $v = r$.
- β denotes assertions involving Opools. Empty and singleton assertions take the similar forms as in Separation Logic. As previously stated, a cell in Opool is an field-value binding of an object (denoted by a reference), thus the singleton assertion takes the form $r_1.a \mapsto r_2$. To make OOSL clear and simple, we do not define $v.a \mapsto \ldots$ as a primitive assertion, because it is not really primitive. Certainly, we can define $v.a \mapsto r$ as $\exists r' \cdot v = r' \wedge r'.a \mapsto r$.

– In addition, we have a special assertion form $\mathsf{obj}(r, T)$ to indicate that r refers to a complete object of type T, and the Opool contains only this object. In Separation Logic, people use $l \mapsto -$ or $l \hookrightarrow\!\mapsto -$ to denote that location l is allocated in current heap. On our side, because the existence of empty object, we cannot use $r.a \mapsto -$ or $r.a \hookrightarrow\!\mapsto -$ to express that the object referred by r is allocated in current Opool. To solve this problem, we introduce the assertion form $\mathsf{obj}(r, T)$ in OOSL. In the follows, we will use $\mathsf{obj}(r, -)$ when we do not care about r's type.

We allow user-defined predicates in OOSL to support user-defined assertions. In fact, user-defined recursive predicates are always necessary to support specification and verification of OO programs involving recursive data structures, e.g., lists, trees, etc. We record these definitions in a *Logic Environment* Λ:

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

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

As a well-formed logic environment, we ask for that Λ must be self-contained, that is: the body assertion ψ of any definition in Λ cannot use symbols not defined in Λ. Further, we require that Λ must be *finite* and *syntactically monotone*[3]. Under these conditions, a fix-point semantics for Λ exists (this technique is standard, like in Description Logic [2]).

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

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

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

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

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

Definition 6 (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 $\mathsf{arity}(\mathcal{I}(p)) = \mathsf{argc}_\Lambda(p)$.* □

[3] For every definition $p(\overline{r}) \doteq \psi$, we require that every symbol occurs in ψ must lie under an even number of negations. This property is named *syntactically monotone*.

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

Fig. 7. Semantic function with interpretation $\mathcal{I}$

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

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

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

We define a semantic function $\mathcal{M} : \mathcal{I} \to \Psi \to \mathbb{P}(\mathrm{State})$ for OOSL, the definition is presented in **Fig.7**, where $\mathcal{M}_{\mathcal{I}}$ means $\mathcal{M}(\mathcal{I})$ in the definitions.

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

Definition 7 (Model of Logic Environment). *Suppose $\mathcal{I}$ is an interpretation of Λ, we say $\mathcal{I}$ is a model of Λ iff for every definition $p(\overline{r}) \doteq \psi$ in Λ, we have:*

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

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

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

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

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

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

Example 1. We give here a simple example to illustrate the idea. Suppose Λ contains only one definition

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

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

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

Taking its limit, we get the standard model that describes all possible lists of a. $\square$

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

Definition 8 (Semantics of Assertions).

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

The OOSL has many interesting features. More details of it can be found in our report [21].