Modular Verification of OO Programs with Interfaces* (Technical Report)

Qiu Zongyan, Hong Ali, and Liu Yijing

LMAM and Department of Informatics, School of Math., Peking University, CHINA
{qzy,hongali,liuyijing}@math.pku.edu.cn

Abstract. *Interface type* in OO languages supports polymorphism, abstraction and information hiding by separating interfaces from their implementations, and thus enhances modularity of programs. However, they cause also challenges to the formal verification. Here we present a study on interface types, and develop a specification and verification theory based on our former VeriJ framework. We suppose multi-specifications for classes inherited from interfaces and the superclass, and keep the verification modular without re-touching the verified codes. The concepts developed in VeriJ, namely the abstract specification and specification predicate, play still important roles in this extension, and thus are proved widely useful and very natural in the formal proofs of OO programs.

1 Introduction

The reliability and correctness for software systems attract more and more attentions, because faults in an important system may cause serious damages or even lose of life. Object-Oriented (OO) techniques are widely used in software practice, and thus the useful techniques supporting high-quality OO development are really demanded, e.g., the formal verification techniques. Core OO features, saying modularity, encapsulation, inheritance, polymorphism, etc., enhance scalability of programs greatly in practice, but they bring also challenges to formal verification. Encapsulation implies information hiding and invisible (and replaceable) implementation details; polymorphism enables dynamic determined behaviors. Both cause difficulties to the verification.

Separating the interface from real implementation is one of the most important techniques in OO development. With this separation, clients can refer to objects of different types via a common interface, and then call methods determined by types of the objects. This enables low coupling of clients from the implementations, and makes the system easily to modify and extend. The technique is used widely, e.g., we can find it in many design patterns [6]. To support this important technique, many OO languages, notably Java and C#, offer special features. In languages without direct support, various features to support the utilization of the interface techniques are available.

To verify OO programs with interfaces raises new challenges. We must define formally the roles played by interfaces, and make clear not only relations among them, but also their connections with the classes. Then we should develop ways for verifying

* Supported by NNSFC Grand No. 90718002 and 61100061.

the implementation classes, as well as verification of the client code which uses the classes via variables of interface types. In addition, we want that the verification could be done modularly: neither re-verifying class implementations, nor touching implementation details of the classes in verifying client code.

Behavioral subtyping [13] is one of most important concepts in OO world, which is a cornerstone in OO practice, as well as an important ingredient of many formal work in OO area, e.g. [2, 9]. An OO program obeying behavioral subtyping gives good support to reason its behavior modularly. However, when multiple interfaces present, how can we think about and define behavioral subtyping without implementations?

Many concepts have been proposed for the verification in OO area, e.g., *model field/abstract field* [3, 11, 14], *data group* [12], and *pure methods* [17], etc. Verifying OO programs with interfaces is also studied in some work, e.g. [16, 14, 7, 2, 5]. [16] integrated interface types and proposed some techniques which inspired many later research. The work on JML and Spec# [7, 2] introduced specifications for interface types with data abstraction to some extent, ensured behavioral subtyping and also developed some verification tools. However, these early work have some avoidable weak points. For example, as pointed by [15], none of these work addresses the inheritance in a satisfactory way, because they either restrict behaviors of subclasses, or require re-verification of inherited methods. In addition, mutable object structures are largely neglected. To remedy the situation, [15, 4] provided one similar idea by suggesting dual specifications for a method, where the *static* one describes its detailed behavior for verifying implementation, and the *dynamic* one supports verifying dynamically bound invocations. However, dual specifications are not necessary, and our VeriJ framework [18] can handle the problem where only one specification for each method is provided.

To develop a verification framework for OO languages with interface types is the goal of this study. It seems not easy to extend the *dual specification* approaches to cover interface types, because interfaces do not have behaviors, neither static nor dynamic. Our VeriJ framework is based on *abstract specifications* and *specification predicates*, while the former supports specifying method behavior on a suitable abstract level, and the later serves to connect the specification with concrete implementations. Based on these concepts, we develop a framework which seems very satisfactory.

The main contributions of this paper are: (1). we give a deep analysis for interface types when the verification is the goal. (2). we define a framework to support abstract specification for interface-based information hiding and encapsulation, and define rules for the verification. We show that the *specification predicate*s play the role to connect abstract specification with implementation details, and support modular verification of different implementing classes under an interface. (3). we propose a general model for dealing with multi-inheritance of specifications, and inference rules for proving the implementation and client code on this sitting modularly. In supporting the verification of Java and C# style interface types, multi-inherited specifications are unavoidable. (4). we developed some examples to show the power and usefulness of our framework. To our limited knowledge, this is the first framework which can avoid reverification of method bodies in a language with rich OO features and interface types.

In the next section, we discuss the crucial situations that a useful theoretical work for interface types must address. We introduce briefly our assertion language and a small

$\textbf{inter}\ I_1\ \{\ T_1\ m(..);\ \ T_2\ f(..);\}\qquad \textbf{inter}\ I_2\ \{\ T_1\ m(..);\ \ T_3\ g(..);\}$
$\textbf{class}\ \ B : \textbf{Object}\ \{\ T_1\ m(..)\{...\}\ \ T_3\ g(..)\{...\}\ \ T_4\ h(..)\{...\}\ \ T_5\ k(..)\{...\}\ \}$
$\textbf{class}\ \ D : B \rhd I_1, I_2\ \{\ T_1\ m(..)\{...\}\ \ T_2\ f(..)\{...\}\ \ T_5\ k(..)\{...\}\ \ T_6\ n(..)\{...\}\ \}$
$\textbf{class}\ \ E : \textbf{Object} \rhd I_1\ \{\ T_1\ m(..)\{...\}\ \ T_2\ f(..)\{...\}\ \ T_7\ p(..)\{...\}\ \}$

Fig. 1. A Program with Interfaces

OO language VeriJ in Section 3, and define its verification framework in Section 4. We illustrate our ideas for modular specification and verification by examples in Section 5, and then discuss some related work and conclude. We have, in the Appendix, some details for OOSL, and some more verification examples.

2 Interfaces and Verification: Basics

To give some ideas for the problem, code in **Fig. 1** is used in the discussion. The basic language we use is similar to Java or JML with some abbreviations for saving space. We will present some issues related to the specification and verification of OO programs with interfaces. We take "type" as a generic word for either a class or an interface.

Code in **Fig. 1** illustrates some cases in programs using interfaces. Here are two interfaces I_1 and I_2, each of which declares some method prototypes. Different interfaces can declare methods with the same name, e.g. m here. Here are also three classes, where class B takes $\textbf{Object}$ as its superclass and implements some methods (with bodies then they are method definitions). B has nothing to do with the interfaces. However, another class D is defined as a subclass of B, which inherits g, h from B, overrides m, k of B, and implements itself new methods f, n. In addition, D implements both interfaces I_1 and I_2, and thus it must implement all methods declared in I_1 and I_2 as required. There are some interesting phenomena: D implements f of I_1 itself, but inherits g from its superclass B to implement g of I_2. The situation for method m is more complex. Here each of interfaces to be implemented has declared a method prototype with name m, in addition, a method with name m is implemented in B too. In D, a new method definition overrides m in B and implements the m in both I_1 and I_2. At last, another class E implements also I_1 with necessary method definitions.

We think that an interface defines a type, while a class defines a type with an implementation. When a class implements some interfaces or inherits a superclass, it defines a subtype of them. To simplify the discussion, we assume that all the data fields in classes are protected, and thus are not visible out of their classes. Under this assumption, a type is just a set of methods with signatures. We must have some type-related constraints. As in Java, when a class implements an interface I, it must provide all implementations for the methods declared in I, by either defining in itself or inheriting from its superclass. When a class implements several interfaces, a method with multiple declarations in these interfaces and/or the superclass, e.g. m in our example code, must have the same signature. Constraints like these should be checked to ensure well-formedness. In the below, we suppose all programs under discussion to be well-formed.

For verification, we need specifications for methods. Assume method m in D has the specification $\pi_{_D}$, then we need to prove that the implementation of m in D satisfies $\pi_{_D}$. In addition, we should support verifications of the client code which calls methods. With interfaces, we need to support verification of programs as:

$$I_1\ x = \mathbf{new}\ D(..);\ I_2\ y = (D)x;\ \ \ldots\ x.m(..);\ y.m(..); \tag{1}$$

Here an object of type D is created and assigned to variable x of type I_1, and then to variable y of type I_2. Afterward, method m is called from x and y, where the D object is used from variables of types I_1 and I_2. To verify the code, we can only refer to information for m in I_1 or I_2, but neither its implementation nor its specification in D. This restriction is clear, because there may be another creation, e.g.,

$$I_1\ z = \mathbf{new}\ E(..);$$

and then the object is passed to the same call statement from x in (1). This means that methods in interfaces must go with their specifications to support verification of client code as (1). The method declarations in interfaces have no body, thus their specifications must be abstract and say nothing about the implementation.

We extend method declarations and definitions in interfaces/classes as follows:

$$T_1\ m(..)\ \langle\varphi\rangle\langle\psi\rangle; \qquad\qquad T_1\ m(..)\ \langle\varphi\rangle\langle\psi\rangle\ \{\ \ldots\ \}$$

where φ and ψ are the pre and post conditions of m, respectively. In I_1 and I_2, we need to specify m based on its parameters and return only, because m has no body here.

To support modular verification, a class C should be not only a behavioral subtype of its superclass, but also a behavioral subtype of its implementing interfaces, because a C object may be used via variables of any of C's supertypes. In our example, D must be a behavioral subtype of B, I_1 and I_2. Suppose the specifications of m in I_1, I_2, and B are π_1, π_2, and $\pi_{_B}$ respectively, then we must prove that m's implementation in D satisfies these three specifications. As a simplification, we may prove some correct relations between $\pi_{_D}$ with each of π_1, π_2, and $\pi_{_B}$.

If a method is explicitly specified, its specification is obvious. We also allow specification inheritance in our framework as in JML and Spec#. This means, when a method is not explicitly specified, its specification is inherited from the supertype(s). Because one class can have more than one supertypes (some interfaces and one superclass), a method may inherit several specifications. For example, if m in D is not explicitly specified, then it inherits specifications π_1, π_2, and $\pi_{_B}$. In this case, we take its specification as a set $\{\pi_1, \pi_2, \pi_{_B}\}$. Then we should define how a method body satisfies a specification as this, and how this specification is used in verifying client code.

Until now, we complete an outline for the problems when we think about the verification of OO programs with interfaces. Based on our previous work VeriJ, with the concepts of abstract specification and specification predication, we can develop a modular verification framework for these OO programs.

3 VeriJ: An OO Language with Specifications

Now we introduce our specification and programming language used in the work.

In [19] we developed OO Separation Logic (OOSL) for describing OO states and reasoning programs. We use it as the assertion language in this work. Here we give a short introduction to OOSL. Readers can refer [19] to find more details.

OOSL is similar to the Separation Logic with some revisions:

$$\begin{array}{l}\rho ::= \mathbf{true} \mid \mathbf{false} \mid r_1 = r_2 \mid r : T \mid r <: T \mid v = r \\ \eta ::= \mathbf{emp} \mid r_1.a \mapsto r_2 \mid \mathsf{obj}(r, T) \\ \psi ::= \rho \mid \eta \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi \Rightarrow \psi \mid \psi * \psi \mid \psi -\!\!* \psi \mid \exists r \cdot \psi \mid \forall r \cdot \psi\end{array}$$

where T is a type, v a variable or constant, r_1, r_2 references, ψ an assertion:

- ρ denotes assertions independent of heaps. For any references r_1 and r_2, $r_1 = r_2$ iff r_1 and r_2 are identical. $r : T$ means that r refers to an object with exact type T. $r <: T$ means that r refers to an object of T or its subtype. And $v = r$ asserts that the value of variable or constant v is r.
- η denotes assertions involving heaps: $\mathbf{emp}$ asserts an empty heap; the singleton assertion $r_1.a \mapsto r_2$ means that the heap is exact a field a of an object (denoted by r_1) holding value r_2; $\mathsf{obj}(r, T)$ means that the heap is exact an entire object of type T, which r refers to. Because the existence of empty objects in OO, we cannot use $r.a \mapsto -$ or $r.a \hookrightarrow -$ to assert the existence of objects in heaps.
- $*$ and $-\!\!*$ are from Separation Logic: $\psi_1 * \psi_2$ means the heap can be split into two parts, where ψ_1 and ψ_2 hold on each part respectively; $\psi_1 -\!\!* \psi_2$ means that if a heap satisfying ψ_1 is added to the heap, the whole heap satisfies ψ_2.

We allow user-defined predicates to extend vocabulary of the assertion language. A predicate definition takes the form $p(\overline{x}) : \psi$, where p is a symbol (predicate name), $\overline{r}$ are its formal parameters, and ψ is the body which is an assertion correlated with $\overline{r}$. Recursive definitions are allowed. In fact, the recursive predicates are indispensable to support specification and verification of programs involving recursive data structures, e.g., lists, trees, etc. Having a definition for symbol p, expression $p(\overline{e})$ can be used as a basic assertion. We use Ψ to denote the set of OOSL assertions.

We use $\psi[v/x]$ (or $\psi[r/x]$, $\psi[r_1/r_2]$) to denote substitutions. We treat $r = v$ the same as $v = r$, and define $v.a \mapsto r$ (which is not a basic assertion here) as $\exists r' \cdot (v = r' \wedge r'.a \mapsto r)$. Some common abbreviations are:

$$r.a \mapsto - \;=\; \exists r' \cdot r.a \mapsto r' \qquad r.a \hookrightarrow r' \;=\; r.a \mapsto r' * \mathbf{true}$$

We use $type(r)$ to denote type of the object which r refers to. Sometimes we need it.

In [19] we defined the semantics for OOSL and proved that most axioms and inference rules for Separation Logic are also correct in OOSL. In practice, we often need to add some mathematical concepts into OOSL, such as relation, set, sequence, etc., to enhance its expressiveness. Such extensions are orthogonal with the core.

We use in this work a small OO language VeriJ which is an extension of a subset of Java with essential OO features. It integrates features of interface, specification and verification with the syntax given in **Fig. 2**, where:

- C and I are class and interface names, respectively. $\mathbf{pub}$ is used to announce that a date field or predicate is publicly accessible. Mutation, field accessing, casting, method invocation, and object creation are all taken as special assignments.

$$
\begin{array}{rcl}
v & ::= & \mathbf{this} \mid x \qquad e ::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid v \mid d \mid e \odot e \\
b & ::= & \mathbf{true} \mid \mathbf{false} \mid e < e \mid e > e \mid e = e \mid \neg b \mid b \vee b \mid b \wedge b \\
c & ::= & \mathbf{skip} \mid x := e \mid v.a := e \mid x := v.a \mid x := (C)v \mid x := v.m(\overline{e}) \mid \\
 & & x := \mathbf{new}\, C(\overline{e}) \mid \mathbf{return}\ e \mid c; c \mid \mathbf{if}\ b\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ c \\
T & ::= & \mathbf{Bool} \mid \mathbf{Object} \mid \mathbf{Int} \mid C \mid I \qquad \pi ::= \langle\varphi\rangle\langle\psi\rangle \qquad M ::= T'\ m(\overline{T_1\ z}) \\
P & ::= & \mathbf{def}\ [\mathbf{pub}]\ p(\mathbf{this}, \overline{a}) \qquad\qquad L ::= \mathbf{inter}\ I\ [: J]\ \{\overline{P};\ \overline{M\ [\pi];}\} \\
K & ::= & \mathbf{class}\ C : B\ [\triangleright \overline{I}]\{\overline{[\mathbf{pub}]\ T\ a;}\ \overline{P : \psi;}\ C(\overline{T_1\ z})\ [\pi]\ \{\overline{T_2\ y};\ c\}\ \ \overline{M\ [\pi]\ \{\overline{T_2\ y};\ c\}}\} \\
G & ::= & (K \mid L) \mid (K \mid L)\, G
\end{array}
$$

Fig. 2. Syntax of VeriJ

- We can have a specification π for a constructor or method in a class or interface. In postcondition ψ we can use $\mathbf{old}(e)$ to denote the value of e in the pre-state. Specifications in a supertype can be inherited or overridden in the subtypes. If a non-overridden method is not explicitly specified, it takes default "$\langle\mathbf{true}\rangle\langle\mathbf{true}\rangle$".
- We have user-defined predicates (*specification predicates* in our words) in the language. P denotes a predicate definition including a body ψ. We ask for that $\mathbf{this}$ is written explicitly as the first parameter to denote the current object. The non-$\mathbf{pub}$ predicates defined in the class can be used in method specification to make it abstract and hide implementation details. In addition, methods declared in interfaces have no implementation, and we often need to declare some predicates and specify the methods based on them. When a class C implements an interface I, it should not only provide implementations for methods declared in I, but also definitions for the predicates declared in I, to connect the method specifications in I (and C) with C's implementations.
- L declares an interface which may inherit another interface J. A class C may implement some interfaces $\overline{I}$, and inherit a class B. A sub-interface should not redeclare the same methods of its super-interfaces. As in Java, each class has a superclass, possibly $\mathbf{Object}$, but may implement zero or more interfaces. We assume all methods are public. For simple we omit method overloading here.

We consider only well-typed programs in verification, and use a static environment to record information in the program text. The environment for program G takes the form $\Gamma_G = (\Delta_G, \Theta_G, \Pi_G, \Phi_G)$, where Δ is the typing environment recording structural information of declarations; Θ is a method lookup environment mapping C, m to its body; Π records method specifications; and Φ records the specification predicates defined in G. We will omit the subscript G when no ambiguity.

As said before, an interface defines a type, and a class defines a type with implementation. We always assume that types in discussion are valid in the program, and use C, D for class names, I for interface name, T for type names, to avoid simple conditions. We use $(T, T') \in \Delta.\mathsf{super}$ to mean that T' is a direct supertype of T. We often omit Δ. In the example of **Fig. 1**, we have $\mathsf{super}(D, I_1)$, $\mathsf{super}(D, I_2)$, and $\mathsf{super}(D, B)$. We use $\mathsf{super}(C)$ to get all supertypes of C. In addition, $C <: T$ means that C is a subtype of T ($<:$ is the transitive and reflective closure of super). We use $\Delta(T)$ to get the map from the method name set of T to their signatures, and then $\Delta(T)(m)$ is the signa-

$$\text{[P-DECLA]}\ \frac{\mathbf{inter}\ I\ \ldots\{\ldots\ \mathbf{def}\ [\mathbf{pub}]\ p(\mathbf{this}, \overline{a}); \ldots\}}{(p(\mathbf{this}, \overline{a}),\ [\mathbf{pub}]\ \mathsf{undef}) \in \Phi(I)}$$

$$\text{[P-DEF]}\ \frac{\mathbf{class}\ C \ldots\ \{\ldots \mathbf{def}\ [\mathbf{pub}]\ p(\mathbf{this}, \overline{a}) : \psi; \ldots\}}{(p(\mathbf{this}, \overline{a}), [\mathbf{pub}]\ \psi) \in \Phi(C)}$$

$$\text{[P-INH-I]}\ \frac{p \text{ not declared in } I, \quad (I, J) \in \mathsf{super}, \quad (p(\mathbf{this}, \overline{a}),\ [\mathbf{pub}]\ \mathsf{undef}) \in \Phi(J)}{(p(\mathbf{this}, \overline{a}),\ [\mathbf{pub}]\ \mathsf{undef}) \in \Phi(I)}$$

$$\text{[P-INH-C]}\ \frac{p \text{ not defined in } C, \quad (C, B) \in \mathsf{super}, \quad (p(\mathbf{this}, \overline{a}), [\mathbf{pub}]\ \psi) \in \Phi(B)}{(p(\mathbf{this}, \overline{a}), [\mathbf{pub}]\ \psi) \in \Phi(C)}$$

$$\text{[S-DEF]}\ \frac{[\mathbf{inter} \mid \mathbf{class}\ ]\ T\ \ldots\{\ldots\ T'\ m(\overline{T_1\ z})\ \langle\varphi\rangle\langle\psi\rangle \ldots\}}{\Pi(T.m) = \langle\varphi\rangle\langle\psi\rangle}$$

$$\text{[S-CON]}\ \frac{\mathbf{class}\ C\ \ldots\{\ldots\ C(\overline{T_1\ z})\ \langle\varphi\rangle\langle\psi\rangle\{\overline{T_2\ y};\ c\} \ldots\}}{\Pi(C.C) = \langle\varphi\rangle\langle\psi\rangle}$$

$$\text{[S-INH1]}\ \frac{\begin{array}{c}\mathbf{class}\ C : B \rhd\ \overline{I}\{\ldots\ T'\ m(\overline{T_1\ z})\ \{\overline{T_2\ y};\ c\} \ldots\},\\ \Pi(\overline{I})(m) = \{\pi_i\}_{i=1}^{k}, \quad \Pi(B.m) = \{\pi_j\}_{j=k+1}^{n}\end{array}}{\Pi(C.m) = \{\pi_i\}_{i=1}^{k} \cup \{\pi_j\}_{j=k+1}^{n}}$$

$$\text{[S-INH2]}\ \frac{\mathbf{class}\ C : B \ldots\{\ldots\}, \quad C \text{ inherits } m \text{ from } B, \quad \Pi(B.m) = \{\pi_i\}_{i=1}^{k}}{\Pi(C.m) = \{\pi_i\}_{i=1}^{k}}$$

Fig. 3. Constructing predicate and specification environments

ture of method m in type T, with the form of $(\overline{T_1\ z}) : T$. We will use $\Delta(T.m)$ as an abbreviation of $\Delta(T)(m)$. The similar abbreviations will be used throughout this paper.

We record all inherited components (fields, method signatures, bodies and specifications, and predicates) for a class as if they are redeclared in the class. Because the basic language supports only single inheritance, all the inherited components are simple to record. A method in a class has a body, either given by a definition, or inherited from the superclass. For a type T, we use $\Theta(T)$ to get the map from method names in T to their bodies. If m is a method name of C, then $\Theta(C.m)$ gets its body. If T is an interface, we suppose $\Theta(T) = \emptyset$. We allow method overridden but not overloading.

For predicates, we record its publicity ($\mathbf{pub}$ or not) with the body, and use $\Phi(T.p)$ to take p's definition in T with the $\mathbf{pub}$ label if existing. For an interface, because there is no implementation, bodies of its predicates are recorded as undef. The specification environment is singular. If method m is defined in class C with specification, its specification is clear, while if it is not explicitly specified, m inherits specification from all C's supertypes. If C inherits m from its superclass, it inherits m's specification in the superclass too. Due to the specification inheritance, when class C implements interfaces $I_1, I_2, \ldots$, and defines method m without giving a specification, m in C may have multiple specifications $\pi_1, \pi_2, \ldots, \pi_k$ if more than one of the interfaces and/or C's superclass have specifications for m. Then, $\Pi(T)$ is a relation from method names to specifications. We write $\langle\varphi\rangle\langle\psi\rangle \in \Pi(T.m)$ when $\langle\varphi\rangle\langle\psi\rangle$ is a specification of m in T, and $\Pi(T.m) = \langle\varphi\rangle\langle\psi\rangle$ when $\langle\varphi\rangle\langle\psi\rangle$ is the only specification.

These components are easy to build by scanning the program text. We give some construction rules of specification predicate and specification environment in **Fig. 3**.

For constructing the predicate environment Φ, [P-DECLA] says that if a specification predicate p is declared with its signature and public modifier in interface I, the

signature and visibility of p and label undef for p's body is recorded for I. Notice that, undef stands here because here the predicate p has not a definition yet. [P-DEF] says that if a definition for predicate p appears in class C, whether it is declared in an implemented interface of C, or firstly defined in C, or already defined in a superclass, this definition of p is recorded for C. [P-INH-I] says if interface I inherits another interface J and p is not declared in I but declared in J, p with its declared information in J is also recorded for I. [P-INH-C] says if C inherits a class B and p is declared in B but not in C, then p'd definition in B is also recorded for C.

For specification environment Π, the constructing rules are also given in **Fig. 3** too. [S-DEF] says if a method is declared either in an interface or class T with the specification $\langle\varphi\rangle\langle\psi\rangle$, we record directly that $\Pi(T.m) = \langle\varphi\rangle\langle\psi\rangle$. [S-CON] says we always record the only specification for the constructor. Rule [S-INH1] says if m is defined in C without a specification, then it inherits the specifications of m from all the supertypes (either interfaces or superclass) of C, if any. Of course, there should be some of them which have m declared. For [S-INH2] deals with the case that C does not have a definition for m but inherits m from its superclass. In this case, it inherits all the specifications of m from its superclass B.

With the environment, the type checking is easy to conduct. We omit it here. One notable fact is that we need also type-checking specifications in VeriJ programs. For a method declaration (or definition) to be well-formed, its pre and post conditions must be well-formed, and the predicates declared in an interface must be realized in its implementation classes, etc. We omit all these details.

4 Verification Framework

Now we develop a framework for modular specification and verification of VeriJ programs. We introduce some notations and definitions in the first.

We use $\Gamma, C, m \vdash \psi$ to denote the fact that assertion ψ holds in method m of class C under Γ. Clearly, here ψ can only be a state-independent assertion. We use $\Gamma, C, m \vdash \{\varphi\}\ c\{\psi\}$ to denote the fact that command c in method m satisfies the specification consisting of precondition φ and postcondition ψ, and $\Gamma \vdash \{\varphi\}\ C.C\{\psi\}$ denotes that constructor C is correct with respect to specification $\langle\varphi\rangle\langle\psi\rangle$ under Γ. For a method, because it may have multiple specifications, we use $\Gamma, C.m \vdash \Pi(C.m)$ to say that $C.m$ is correct wrt. its every specification. We write $\Gamma \vdash \{\varphi\}\ C.m\{\psi\}$ to state $C.m$ is correct wrt. specification $\langle\varphi\rangle\langle\psi\rangle$.

For OO programs, behavioral subtyping is crucial in modular verification. To introduce it into our framework, we define a refinement relation between specifications.

Definition 1 (Refinement of Specification). *Given two specifications $\langle\varphi_1\rangle\langle\psi_1\rangle$ and $\langle\varphi_2\rangle\langle\psi_2\rangle$, we say that the latter one refines the former in context Γ, C,* iff *there exists an assertion R which is free of the program variables, such that $\Gamma, C \vdash (\varphi_1 \Rightarrow \varphi_2 * R) \wedge (\psi_2 * R \Rightarrow \psi_1)$. We use $\Gamma, C \vdash \langle\varphi_1\rangle\langle\psi_1\rangle \sqsubseteq \langle\varphi_2\rangle\langle\psi_2\rangle$ to denote this fact.* □

Liskov [13] defined the condition for specification refinement as $\varphi_1 \Rightarrow \varphi_2 \wedge \psi_2 \Rightarrow \psi_1$. Above definition is its extension by taking the storage extension into account. This definition follows also the *nature refinement order* proposed in [8].

$$\text{[H-THIS]}\ \Gamma, T, m \vdash \mathbf{this} : T \qquad \text{[H-SKIP]}\ \Gamma \vdash \{\varphi\}\ \mathbf{skip}\{\varphi\} \qquad \text{[H-ASN]}\ \Gamma \vdash \{\varphi[e/x]\}\ x := e; \{\varphi\}$$

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

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

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

$$\text{[H-SEQ]}\ \frac{\Gamma \vdash \{\varphi\}\ c_1\{\psi\},\ \Gamma \vdash \{\psi\}\ c_2\{R\}}{\Gamma \vdash \{\varphi\}\ c_1\ c_2\{R\}} \qquad \text{[H-COND]}\ \frac{\Gamma \vdash \{b \wedge \varphi\}\ c_1\{\psi\},\ \Gamma \vdash \{\neg b \wedge \varphi\}\ c_2\{\psi\}}{\Gamma \vdash \{\varphi\}\ \mathbf{if}\ b\ c_1\ \mathbf{else}\ c_2\{\psi\}}$$

$$\text{[H-ITER]}\ \frac{\Gamma \vdash \{b \wedge I\}\ c\{I\}}{\Gamma \vdash \{I\}\ \mathbf{while}\ b\ c\{\neg b \wedge I\}} \qquad \text{[H-FRAME]}\ \frac{\Gamma, C, m \vdash \{\varphi\}\ c\{\psi\} \quad \mathsf{FV}(R) \cap \mathsf{MD}(c) = \emptyset}{\Gamma, C, m \vdash \{\varphi * R\}\ c\{\psi * R\}}$$

$$\text{[H-CONS]}\ \frac{\Gamma, C, m \vdash \varphi \Rightarrow \varphi', \quad \Gamma, C \vdash \psi' \Rightarrow \psi \quad \Gamma, C, m \vdash \{\varphi'\}\ c\{\psi'\}}{\Gamma, C, m \vdash \{\varphi\}\ c\{\psi\}} \qquad \text{[H-EX]}\ \frac{\Gamma, C, m \vdash \{\varphi\}\ c\{\psi\} \quad r \text{ is free in } \varphi, \psi}{\Gamma, C, m \vdash \{\exists r \cdot \varphi\}\ c\{\exists r \cdot \psi\}}$$

$$\text{[H-OLD]}\ \frac{\forall \langle\varphi\rangle\langle\psi\rangle \in \Pi(T.m) \bullet \Gamma, T, m \vdash (\overline{z = r} \wedge \varphi[\overline{r}/\overline{z}]) \Rightarrow \psi'}{\Gamma, T, m \vdash \psi'[\mathbf{old}(e)/e]}$$

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

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

Fig. 4. Basic Inference Rules

Interfaces, their inheritance, and the behavioral subtyping relation also correlate with verifications, which we call *static verification*. For an interface I with super-interface I', if method m in I has a new specification $\langle\varphi\rangle\langle\psi\rangle$ overriding its original $\langle\varphi'\rangle\langle\psi'\rangle$ in I', we must verify refinement relation $\Gamma, I \vdash \langle\varphi'\rangle\langle\psi'\rangle \sqsubseteq \langle\varphi\rangle\langle\psi\rangle$. This verification is done only on the logic level, because of no method body in interfaces. This verification is supported by the abstract specifications in our framework.

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

Definition 2 (Correct Program). *Program G is* correct, iff *for each $\langle\varphi\rangle\langle\psi\rangle \in \Pi_G(T.m)$ (and $\Pi_G(C.C) = \langle\varphi\rangle\langle\psi\rangle$), we have $\Gamma_G \vdash \{\varphi\}\, T.m\{\psi\}$ (and $\Gamma_G \vdash \{\varphi\}\, C.C\{\psi\}$).* □

Basic inference rules are given in **Fig. 4**. Rules for assignment, mutation and lookup are similar as their counterparts in Separation Logic. [H-CAST] is special for type casting, and [H-RET] is similar to [H-ASN] but the target is res specially. Rules for composition structures, and rules [H-CONS], [H-EX] take the same forms as what in Hoare logic. [H-FRAME] is the important frame rule, where $\mathsf{FV}(R)$ is the set of all program variables (including internal res) in assertion R, and $\mathsf{MD}(c)$ is the set of variables modified by command c. [H-THIS] is simply a type assertion. [H-OLD] says that if assertion ψ' is provable in the pre-state, then $\psi'[\mathbf{old}(e)/e]$ is provable in the body of the method. A similar rule for constructors is omitted here. Note that here $\forall$ is used only as a shorthand but not a quantifier in logic. Similar notations are used below.

Rules [H-DPRE], [H-SPRE] are key to show our idea that non-public specification predicates have their scopes, and thus can have more than one definitions in the classes crossing the class hierarchy, to implement polymorphism. If a predicate invoked is in scope (in its class or the subclasses), it can be unfolded to its definition. These rules support hiding implementation details, even the details are in the definition of the predicates used in method specifications. However, these two rules are different. [H-DPRE] says if r is of the type D, then in any subclass of D, $p(r, \overline{r'})$ can be unfolded to the body of p. [H-SPRE] is for the static binding, in that case, $D.p(r, \overline{r'})$ is unfolded to its definition in D, where $\mathsf{fix}(D, \psi)$ gives the *instantiation* of ψ in D:

$$\mathsf{fix}(D,\psi) = \begin{cases} \neg\mathsf{fix}(D,\psi'), & \text{if } \psi \text{ is } \neg\psi' \\ \mathsf{fix}(D,\psi_1) \otimes \mathsf{fix}(D,\psi_2), & \text{if } \psi \text{ is } \psi_1 \otimes \psi_2 \\ \exists r \cdot \mathsf{fix}(D,\psi'), & \text{if } \psi \text{ is } \exists r \cdot \psi' \\ D.q(\mathbf{this}, \overline{r}), & \text{if } \psi \text{ is } q(\mathbf{this}, \overline{r}) \\ \psi, & \text{otherwise.} \end{cases}$$

Here $\otimes$ can be $\vee$, $*$, or $-\!\!*$. Intuitively, fix substitutes predicate names with their definitions in D to their complete names, and then uses the resulting assertion, so it fixes the meaning of an assertion with respect to D. In other words, this function provides a static and fixed explanation for ψ according to a given class. Notice here in unfolding $D.q(r, \overline{r'})$, we use $\mathsf{fix}(D, \psi)$ to fix the meaning of q at first, then do the substitution. With this definition, we can have the correct expansion, and at the same time, avoid infinite expansion in unfolding the recursive defined predicate.

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

Rules related to methods and constructors are given in **Fig. 5** where there is a default side-condition that local variables $\overline{y}$ are not free in φ, ψ. This can be provided by renaming when necessary. An available method in class C can have a specification in C, thus have a definition, or have no specification in C but might be a definition, or an inherited definition with also inherited specification from its superclass. Therefore, we define rules according to these three cases for verifying methods.

[H-MTHD1] is for verifying methods with a specification (and of course a definition) in a class. The rule demands firstly that $C.m$'s body meets its specification, and then asks to check the refinement relations between specifications of m in C and C's supertypes, if existed. Here we promote Π to type set, thus $\Pi(\mathsf{super}(C))(m)$ gives specifications for m in C's supertypes. If there are such specifications, we have to prove the refinement relation with each of them. If there is no, this check is true by default.

[H-MTHD2] is for verifying methods defined in classes without specification. Taking m of C as an example, in this case, we need to verify that the body of m implements correctly with every specification of m in C's supertypes, because C inherits all these specifications, and m may be called from variables of these types. Here $\Pi(C.m)$ is the same set as if we took the type set $\mathsf{super}(C)$ then took all specifications of m from these types. Now we do not need to prove specification refinement relation anymore even if some predicates used in the specifications have been overridden in C. After we have verified m's new body with its each specification in $\Pi(C.m)$, the abstract specifications seem to equivalence from view of the clients.

$$\text{[H-MTHD1]}\ \frac{\begin{array}{c} C \text{ has specification for } m, \quad \Theta(C.m) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\}, \quad \Pi(C.m) = \langle\varphi\rangle\langle\psi\rangle \\ \Gamma, C, m \vdash \{\mathbf{this} : C \wedge \overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \varphi[\overline{r}/\overline{z}]\}\ c\{\psi[\overline{r}/\overline{z}]\} \\ \forall\, \langle\varphi'\rangle\langle\psi'\rangle \in \Pi(\mathsf{super}(C))(m) \bullet \Gamma, C \vdash \langle\varphi'\rangle\langle\psi'\rangle \sqsubseteq \langle\varphi\rangle\langle\psi\rangle \end{array}}{\Gamma \vdash \{\varphi\}\ C.m\{\psi\}}$$

$$\text{[H-MTHD2]}\ \frac{\begin{array}{c} C \text{ defines } m \text{ without specification}, \quad \Theta(C.m) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \forall\, \langle\varphi\rangle\langle\psi\rangle \in \Pi(C.m) \bullet \Gamma, C, m \vdash \{\mathbf{this} : C \wedge \overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \varphi[\overline{r}/\overline{z}]\}\ c\{\psi[\overline{r}/\overline{z}]\} \end{array}}{\Gamma, C.m \vdash \Pi(C.m)}$$

$$\text{[H-MINH]}\ \frac{\begin{array}{c} C \text{ inherits } D.m, \quad \forall\, \langle\varphi\rangle\langle\psi\rangle \in \Pi(C.m) \bullet \Gamma, C \vdash \langle\varphi\rangle\langle\psi\rangle \sqsubseteq \langle\mathsf{fix}(D,\varphi)\rangle\langle\mathsf{fix}(D,\psi)\rangle \\ \forall\, I \in \mathsf{super}(C) \wedge \Pi(I.m) = \langle\varphi'\rangle\langle\psi'\rangle \bullet \Gamma, C \vdash \langle\varphi'\rangle\langle\psi'\rangle \sqsubseteq \Pi(C.m) \end{array}}{\Gamma, C.m \vdash \Pi(C.m)}$$

$$\text{[H-CONSTR]}\ \frac{\begin{array}{c} \Pi(C.C) = \langle\varphi\rangle\langle\psi\rangle, \quad \Theta(C.C) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \Gamma, C, C \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}}\ \wedge \mathsf{raw}(\mathbf{this}, C) * \varphi[\overline{r}/\overline{z}]\}\ c\{\psi[\overline{r}/\overline{z}]\} \end{array}}{\Gamma \vdash \{\varphi\}\ C.C\{\psi\}}$$

$$\text{[H-INV]}\ \frac{\Gamma, C, m \vdash v : T, \quad \langle\varphi\rangle\langle\psi\rangle \in \Pi(T.n)}{\Gamma, C, m \vdash \{v = r \wedge \overline{e = r'} \wedge \varphi[r, \overline{r'}/\mathbf{this}, \overline{z}]\}\ x := v.n(\overline{e})\ \{\psi[r, \overline{r'}, x/\mathbf{this}, \overline{z}, \mathsf{res}]\}}$$

$$\text{[H-NEW]}\ \frac{\Pi(C'.C') = \langle\varphi\rangle\langle\psi\rangle}{\Gamma, C, m \vdash \{\overline{e = r'} \wedge \varphi[\overline{r'}/\overline{z}]\}\ x := \mathbf{new}\, C'(\overline{e})\{\exists r \cdot x = r \wedge \psi[r, \overline{r'}/\mathbf{this}, \overline{z}]\}}$$

Fig. 5. Inference Rules related to Methods and Constructors

[H-MINH] is for verifying inherited methods. The rule asks specially to check if m's specification(s) (inheriting from D, maybe more than one because D might inherit some specifications from its supertype(s)) interpreted in C is compatible with its interpretation in D. Here we use $\mathsf{fix}(D, \bullet)$ to fix the meaning of predicates. In addition, we check if the method satisfies each specification of m in C's implementing interfaces (if any) by proving the refinement relation. Here $\pi \sqsubseteq \{\pi_i\}_i$ is defined as $\exists i \cdot \pi \sqsubseteq \pi_i$.

Rule [H-CONSTR] is for constructors which has a similar form with [H-MTHD1]. However, a constructor will not have multiple specifications. Here $\mathsf{raw}(\mathbf{this}, C)$ specifies that $\mathbf{this}$ refers to a newly created raw object of type C, and then c modifies its state. The definition of $\mathsf{raw}(r, N)$ is

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

Last two rules are for method invocation and object creation. Note that $T.n$ may have multiple specifications, and we can use any of them in proving client code. Due to the *behavioral subtyping*, it is enough to do the verification by the type of variable v. Because [H-INV] refers to only specifications, recursive methods are supported.

The soundness of these rules are easy to prove. However, readers may think that [H-MTHD2] is not very satisfactory because it asks for verifying method body for possibly several times. The first answer is that this is necessary, because different specifications for m in the superclass or implemented interfaces may cover different aspects of m's behavior. The definition of m in subclass C must satisfy each of these specifications. However, we may give a new (and weaker) rule to avoid some method body

verifications, if we can find that a specification $\langle\varphi\rangle\langle\psi\rangle$ is the strongest:

$$\text{[H-MTHD2']}\ \frac{\begin{array}{r} C \text{ defines } m \text{ without specification},\ \ \Theta(C.m) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \exists\, \langle\varphi\rangle\langle\psi\rangle \in \Pi(C.m) \bullet ((\forall\, \langle\varphi'\rangle\langle\psi'\rangle \in \Pi(C.m) \bullet \Gamma, C \vdash \langle\varphi'\rangle\langle\psi'\rangle \sqsubseteq \langle\varphi\rangle\langle\psi\rangle) \\ \wedge\ \Gamma, C, m \vdash \{\mathbf{this} : C \wedge \overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \varphi[\overline{r}/\overline{z}]\}\ c\{\psi[\overline{r}/\overline{z}]\}) \end{array}}{\Gamma, C.m \vdash \Pi(C.m)}$$

Here $\langle\varphi\rangle\langle\psi\rangle$ is the strongest one which refines all the other specifications, including itself. However, if there is no strongest one in $\Pi(C.m)$, this rule will be not applicable, even the method body in C does satisfy all the specifications. In addition, based on the general [H-MTHD2], we may develop some other rules in advance.

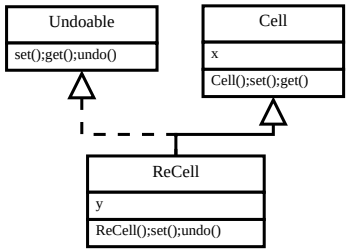
Here we see how the information given by developers affects the verification. A given specification for a method is a specific requirement and induces some special proof obligations. It forms a connection between the implementation with the surrounding world: the implemented interfaces, the superclass, and the client codes. When no given specification, we will need to verify more to ensure all the possibilities.

Clearly, our verification framework is modular, because we do not need to re-touch the code in superclass when consider a subclass. In adding a new class to existed code, we just need to verify the new class but not re-consider the existing part. On the other side, in verifying client code, no matter the method invocation is from a variable of a class or an interface, we do not need to consider the real object the variable refers to. This shows that our basic framework, based on the abstract specification and specification predicate concepts, can be naturally extended to support the verification of programs with interfaces. From this extension, we see our framework is nature in dealing with wide spectrum of OO programs. To reveal these good properties, we consider an examples in the next section. More examples can be found in the Appendix.

5 An Example

The hierarchy of classes used here is given in **Fig. 6** with a UML class diagram. It is a variant of a typical example used in OO program's specification and verification study. A class $Cell$ offers simple methods to set and get value of its field x. Then we want to have some class which can roll back one previous value, and thus declare an interface $Undoable$ with one more method $undo$ than $Cell$. To offer a real class, we define $ReCell$ which inherits $Cell$ and implements $Undoable$. This subclass contains a new field y for saving the old value of x when it is set. To implement the methods declared in $Undoable$, class $ReCell$ defines a method $undo$, inherits $Cell.get$, and overrides $Cell.set$ by a new definition. Here we omit the constructors for simple.

In **Fig. 6** we give also specifications for some methods, in which some specification predicates are used to hide internal information of classes from its clients. Class $Cell$ defines predicate $cell(\mathbf{this}, v)$ to denote that the value of x is v, in $ReCell$ this predicate is overridden by a new definition. Interface $Undoable$ defines predicates $cell$ and bak to assert the current and backup values. Implementations of these predicates is left to its implementing class. Note that predicate $ReCell.cell(\mathbf{this}, v)$ records both current value v in x and some unconcerned value for y, and $ReCell.bak(\mathbf{this}, v)$ records



$\mathbf{class}\ \mathit{Cell} : \mathbf{Object}\{$
$\quad \mathbf{Int}\ x;$
$\quad \mathbf{def}\ \mathit{cell}(\mathbf{this}, v) : \mathbf{this}.x \hookrightarrow v;$
$\quad \mathbf{void}\ \mathit{set}(\mathbf{Int}\ v)$
$\quad \langle \mathit{cell}(\mathbf{this}, -)\rangle\langle \mathit{cell}(\mathbf{this}, v)\rangle$
$\quad \{\ \mathbf{this}.x = v;\ \}$
$\quad \mathbf{Int}\ \mathit{get}()\ \langle \mathit{cell}(\mathbf{this}, v)\rangle\langle \mathsf{res} = v\rangle$
$\quad \{\ \mathbf{Int}\ c;\ c = \mathbf{this}.x;\ \mathbf{return}\ \ c;\}$
$\}$

$\mathbf{inter}\,\mathit{Undoable}\{$
$\quad \mathbf{def}\ \mathit{cell}(\mathbf{this}, v);$
$\quad \mathbf{def}\ \mathit{bak}(\mathbf{this}, v);$
$\quad \mathbf{void}\ \mathit{set}(\mathbf{Int}\ v)\ \langle \mathit{cell}(\mathbf{this}, b)\rangle$
$\quad\quad \langle \mathit{cell}(\mathbf{this}, v) \wedge \mathit{bak}(\mathbf{this}, b)\rangle;$
$\quad \mathbf{Int}\ \mathit{get}()\ \langle \mathit{cell}(\mathbf{this}, v)\rangle\langle \mathsf{res} = v\rangle;$
$\quad \mathbf{void}\ \mathit{undo}()\ \langle \mathit{bak}(\mathbf{this}, b)\rangle\langle \mathit{cell}(\mathbf{this}, b)\rangle;$
$\}$

$\mathbf{class}\ \mathit{ReCell} : \mathit{Cell} \triangleright \mathit{Undoable}\{$
$\quad \mathbf{Int}\ y;$
$\quad \mathbf{def}\ \mathit{cell}(\mathbf{this}, v) : \mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow -;$
$\quad \mathbf{def}\ \mathit{bak}(\mathbf{this}, v) : \mathbf{this}.x \hookrightarrow - * \mathbf{this}.y \hookrightarrow v;$
$\quad \mathbf{void}\ \mathit{set}(\mathbf{Int}\ v)\ \{\ \mathbf{Int}\ c;$
$\quad\quad c = \mathbf{this}.x;\ \mathbf{this}.y = c;\ \mathbf{this}.x = v;\}$
$\quad \mathbf{void}\ \mathit{undo}()\ \{\ \mathbf{Int}\ c;$
$\quad\quad c = \mathbf{this}.y;\ \mathbf{this}.x = c;\}$
$\}$

Fig. 6. Codes and specifications of interface $\mathit{Undoable}$ and classes Cell, ReCell

Proving $\mathit{Cell.set}$**:**
$\{\mathit{cell}(\mathbf{this}, -)\}$
$\{\mathbf{this}.x \hookrightarrow -\}$
$\mathbf{this}.x = v;$
$\{\mathbf{this}.x \hookrightarrow v\}$

Proving $\mathit{Cell.get}$**:**
$\{\mathit{cell}(\mathbf{this}, v) \wedge c = 0\}$
$\{\mathbf{this}.x \hookrightarrow v \wedge c = 0\}$
$c = \mathbf{this}.x;$
$\{\mathbf{this}.x \hookrightarrow v \wedge c = v\}$
$\mathbf{return}\ \ c;$
$\{\mathsf{res} = v\}$

Proving $\mathit{ReCell.undo}$**:**
$\{\mathit{bak}(\mathbf{this}, b) \wedge c = 0\}$
$\{c = 0 \wedge \mathbf{this}.x \hookrightarrow - * \mathbf{this}.y \hookrightarrow b\}$
$c = \mathbf{this}.y;\ \ \mathbf{this}.x = c;$
$\{c = b \wedge \mathbf{this}.x \hookrightarrow c * \mathbf{this}.y \hookrightarrow b\}$
$\{\mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow b\}$
$\{\mathit{cell}(\mathbf{this}, b) \wedge \mathit{bak}(\mathbf{this}, b)\}$
$\{\mathit{cell}(\mathbf{this}, b)\}$

Fig. 7. Proofs for Some Simple Cases

v as the value of y and leaves value of x unconcerned. Because $\mathit{ReCell.set}$ is not explicitly specified, it inherits two specifications from $\mathit{Undoable.set}$ and $\mathit{Cell.set}$. In the same way, $\mathit{ReCell.get}$ inherits a specification from $\mathit{Cell.get}$; and $\mathit{ReCell.undo}$ inherits a specification from $\mathit{Undoable.undo}$. All of these are recorded in the specification environment, and can be used for verifying the program.

We need to prove that the code is correct by the related inference rules given in Section 4 before give the code to clients. Due to the page limited, we give here only main part of the proof to illustrate the usage of our framework. It is clear that, to verify methods in Cell, we should use rule [H-MTHD1] because all of the methods are defined and specified in Cell. Here the premise for specification refinement is vain and thus is trivially true. We only need to verify the method bodies. The two proofs are simple and given in **Fig. 7**.

For $\mathit{ReCell.set}$ and $\mathit{ReCell.undo}$, we need to prove that their bodies meet the respectively inherited specifications from the supertypes of ReCell, because these methods are not specified explicitly. This situation asks us to use rule [H-MTHD2]. The proof for $\mathit{ReCell.undo}$ is simple and given also in **Fig. 7**. For $\mathit{ReCell.set}$, we need

Proving $ReCell.set$ **with** $\Pi(Undoable.set)$**:**	**Proving** $ReCell.set$ **with** $\Pi(Cell.set)$**:**
$\{cell(\mathbf{this}, b) \wedge c = 0\}$	$\{cell(\mathbf{this}, -) \wedge c = 0\}$
$\{c = 0 \wedge \mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow -\}$	$\{c = 0 \wedge \mathbf{this}.x \hookrightarrow -\}$
$c = \mathbf{this}.x;\ \mathbf{this}.y = c;$	$\{c = 0 \wedge \exists b \cdot \mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow -\}$
$\{c = b \wedge \mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow c\}$	$c = \mathbf{this}.x;\ \mathbf{this}.y = c;$
$\mathbf{this}.x = v;$	$\{\exists\, b \cdot c = b \wedge \mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow c\}$
$\{\mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow b\}$	$\mathbf{this}.x = v;$
$\{(\mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow -) \wedge$	$\{\exists\, b \cdot \mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow b\}$
$\quad(\mathbf{this}.x \hookrightarrow - * \mathbf{this}.y \hookrightarrow b)\}$	$\{\mathbf{this}.x \hookrightarrow v\}$
$\{cell(\mathbf{this}, v) \wedge bak(\mathbf{this}, b)\}$	$\{cell(\mathbf{this}, v)\}$

Fig. 8. Proofs for $ReCell.set$ with two specifications

to prove that it meets its two inherited specifications from $Undoable$ and $Cell$ firstly. The proofs are given in **Fig. 8**. Based on these proofs, we can easily conclude that $ReCell.set$ meets its specification.

In addition, we find that rule [H-MTHD2'] can also be used here to avoid verifying the method body more times because there exists a refinement relation between the specifications. We show the proof here as an example. To prove in this way, due to we have proved that $ReCell.set$ meets its specification inherited from $Undoable$, now we need only check the refinement relation. By **Definition 1**, we have trivially:

$$
\begin{array}{l}
\Gamma, ReCell \vdash \\
\langle cell(\mathbf{this}, b)\rangle\langle cell(\mathbf{this}, v) \wedge bak(\mathbf{this}, b)\rangle \Rightarrow \langle cell(\mathbf{this}, -)\rangle\langle cell(\mathbf{this}, v)\rangle \\
\Rightarrow \\
\langle cell(\mathbf{this}, -)\rangle\langle cell(\mathbf{this}, v)\rangle \sqsubseteq \langle cell(\mathbf{this}, b)\rangle\langle cell(\mathbf{this}, v) \wedge bak(\mathbf{this}, b)\rangle
\end{array}
$$

This derivation tells us "$\Gamma, ReCell \vdash \Pi(Cell.set) \sqsubseteq \Pi(Undoable.set)$". Thus, the body of $ReCell.set$ also meets the inherited specification from its superclass $Cell$ according to our weaker rule [H-MTHD2'].

For $ReCell.get$, rule [H-MINH] asks us to prove only specification refinement relations, thus we avoid re-verifying method body and achieve modular verification. The specification of $ReCell.get$ is inherited from $Cell.get$, which is a single specification. Thus, the refinement relation between specifications of $Undoable.get$ and $ReCell.get$ is "$\Gamma, ReCell \vdash \langle\varphi'\rangle\langle\psi'\rangle \sqsubseteq \langle\varphi\rangle\langle\psi\rangle$", where $\varphi = \varphi' = cell(\mathbf{this}, v)$ and $\psi = \psi' = (\mathsf{res} = v)$. This relation holds trivially.

For proving the specification refinement relation between $Cell.get$ and $ReCell.get$, the case is different, that is to prove

$$\Gamma, ReCell \vdash \langle cell(\mathbf{this}, v)\rangle\langle \mathsf{res} = v\rangle \sqsubseteq \langle \mathsf{fix}(Cell, cell(\mathbf{this}, v))\rangle\langle \mathsf{fix}(Cell, \mathsf{res} = v)\rangle$$

By the **Definition 1** and the definition of fix, we need to prove that there exists an assertion R such that

$$
\begin{array}{l}
\Gamma, ReCell \vdash \\
(cell(\mathbf{this}, v) \Rightarrow \mathsf{fix}(Cell, cell(\mathbf{this}, v)) * R) \wedge (\mathsf{fix}(Cell, \mathsf{res} = v) * R \Rightarrow (\mathsf{res} = v)) \\
\Rightarrow \\
(cell(\mathbf{this}, v) \Rightarrow Cell.cell(\mathbf{this}, v) * R) \wedge (Cell.(\mathsf{res} = v) * R \Rightarrow (\mathsf{res} = v)) \\
\Rightarrow \\
((\mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow -) \Rightarrow (\mathbf{this}.x \hookrightarrow v * R)) \wedge ((\mathsf{res} = v) \Rightarrow (\mathsf{res} = v))
\end{array}
$$

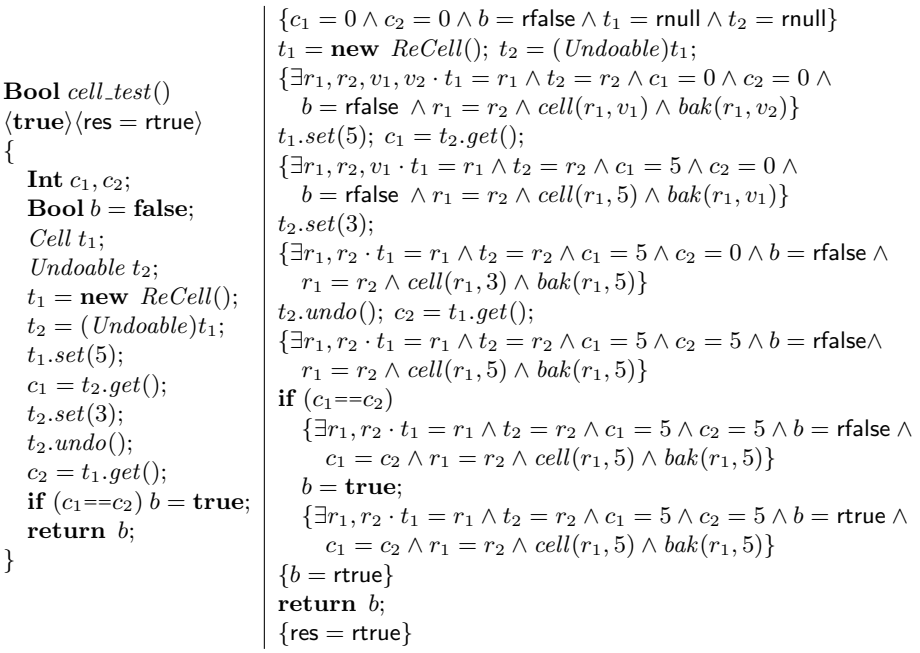


Fig. 9. A Client Method and Its Proof

Let "$R = \mathbf{this}.y \hookrightarrow -$", then we have the above implications true easily. Therefore, we can conclude that $ReCell.get$ meets its specification.

Now we show how a client code can be verified by just referring to the specifications in interfaces and classes, thus is done abstractly and modularly. In **Fig. 9** (left), we define a method $cell\_test$ which declares a variable of type $Cell$ but actually assigns it an object of $ReCell$. Then a new variable t_2 is declared and assigned the same object by casting t_1 to $Undoable$. We give the proof of this method in detail in the figure too. The proof involves only the abstract specifications of the interface and classes.

6 Related Work and Conclusion

To support specifications and verification of OO programs with interface types, we develop here a formal framework which offers modularity for both specification and verification. The OO language VeriJ used here takes the pure reference semantics. A version of Separation Logic, named OOSL, is used for specifying and reasoning VeriJ programs. We suggest *abstract specifications* for describe behaviors of methods. This technique can support also "behavioral" specification for the method declarations in interfaces which have no implementations. We introduce *specification predicates* to link abstract specifications with implementation details, which serve also the connection between the classes with the interfaces which they implement.

We design rules for visibility, inheritance and overriding of specification predicates and method specifications, and develop a set of inference rules which can derive proof obligations from program with specifications for verifying VeriJ programs. Our approach supports full encapsulation for the implementation details, and can also avoid re-verification of inherited methods. In contrast to the work presented in [15, 4], we use only one specification for each method. As in the main-stream OO languages, e.g. Java and C#, here one class may implement several interfaces, as well as inherits a superclass. Our framework support inheriting multi-specifications for methods, and we propose inference rules for proving programs in this situation. By an example, with more examples in **Appendix B**, we show that the framework can dealing with various common problems encountered in OO practice.

The research on the specification and verification in JML and Spec# frameworks [9, 7, 10, 2, 1] considered also interface types. Similarly, these frameworks support method specifications in interfaces, and allow specification inheritance. The refinement relation between supertypes and their subtypes are defined to pursue modular reasoning and behavioral subtyping. Differently, Spec# requires overriding methods in a subtype inherit the same preconditions from its supertypes while postconditions can be strengthened. This brings a big constraint on implementations in subtypes. Actually, to allow more flexible behaviors, we should permit not only strengthening postconditions but also weakening preconditions in subtypes, as what we and JML do. Notably, our framework is more general. In one side, we develop an approach for the inheritance with multiple specifications for methods. In addition, our definition for specification refinement allows storage extension of subclass, which is necessary for dealing with mutable OO structures but totally omitted in Spec# and JML frameworks.

The abstraction techniques adopted in JML and Spec# are similar. Both use model fields and calls of pure methods in their specifications. We find that such calls in specification is not abstract and convenient enough for clients to use and understand. Because it may enforce clients to know about what these pure methods do. We propose specification predicates to hide information from clients and use them in specifications in interfaces (and classes) to provide enough information for verifying class and client codes conveniently by suitable verification rules. In addition, as pointed by [15], the early work, including JML and Spec#, can not avoid re-verification of the inherited methods. That might be another weakness of the approaches based on the model fields and pure methods etc.

In this work, we utilize structures in programs and specification predicates, as the semantic link over the class hierarchy, rather than only linking the *abstract predicate families* [15] to classes by the type of their first parameter and a tag. Use only one specification for a method, we can get rid of repeated expressions, and express the semantic decision for the class design only in the local defined predicates. This feature makes it better to support the *single point rule* in the specifications of programs, which is extremely important in programming practice. In addition, it is not clear how the abstract predicate families and dual specifications mechanisms can be used (extended) to support specification and verification of OO programs with interface types. By successfully extend our framework to support the interface features, we see more clearly the usefulness of the concept specification predicates and its potential power. In fact, the

key point of our approach is to introduce polymorphism concepts into the specification and verification framework, that is learnt from the successful OO practice.

As the future work, we will explore further the potentials of our approach, to support more OO and verification features, such as object invariants, frame problems and confinement, open programs, and so on.

References

1. M. Barnett, M. Fähndrich, K.R.M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
3. 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.
4. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. In *POPL '08*, pages 87–99, New York, NY, USA, 2008. ACM.
5. D. Distefano and M.J. Parkinson J. jstar: Towards practical verification for java. *ACM SIGPLAN Notices*, 43(10):213–226, 2008.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addlison Wesley, 1994.
7. G. Leavens. JML's rich, inherited specifications for behavioral subtypes. *Formal Methods and Software Engineering*, pages 2–34, 2006.
8. 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, 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 and P. Müller. Information hiding and visibility in interface specifications. In *Software Engineering*.
11. K. R. M. Leino. *Toward reliable modular programs*. PhD thesis, California Institute of Technology, Pasadena, CA, USA, 1995. UMI Order No. GAX95-26835.
12. K. R. M. Leino. Data groups: specifying the modification of extended state. *SIGPLAN Notices*, 33:144–153, 1998.
13. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programing Languages and Systems*, 16(6):1811–1841, 1994.
14. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2000.
15. Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75–86, New York, NY, USA, 2008. ACM.
16. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. *Technische Universität München*, 1997.
17. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
18. Liu Yijing, Hong Ali, and Qiu Zongyan. Inheritance and modularity in specification and verification of OO programs. In *TASE 2011*, pages 19–26. IEEE Computer Society, 2011.
19. Liu Yijing and Qiu Zongyan. A separation logic for OO programs. In *FACS 2010*, volume 6921 of *LNCS*, pages 88–105. Springer, 2011.

A OOSL: Some Details and Its Semantics

Here we give some details about OOSL. A complete treatment can be found in [19], including some properties of OOSL, and a careful comparison with some related works.

We use a revised Stack-Heap state model for OO programs. A state $s = (\sigma, O) \in \mathrm{State}$ consists of a store and a heap:

$$\mathrm{Store} \mathrel{\widehat{=}} \mathrm{Name} \rightarrow_{\mathsf{fin}} \mathrm{Ref}, \quad \mathrm{Heap} \mathrel{\widehat{=}} \mathrm{Ref} \rightarrow_{\mathsf{fin}} \mathrm{Name} \rightarrow_{\mathsf{fin}} \mathrm{Ref}$$
$$\mathrm{State} \;\widehat{=}\; \mathrm{Store} \times \mathrm{Heap}$$

Name is an infinite set of names, with specially $\mathbf{true}$, $\mathbf{false}$, $\mathbf{null}$ for boolean constants and null value respectively. Type is an infinite set of types, where $\mathbf{Object}, \mathbf{Null}, \mathbf{Bool}, \mathbf{Int} \in \mathrm{Type}$. Ref is an infinite set of references, i.e., object identities. It contains three constants: rtrue, rfalse refer to the two $\mathbf{Bool}$ objects, and rnull. For any $\sigma \in \mathrm{Store}$, we assume $\sigma\mathbf{true} = \mathsf{rtrue}$, $\sigma\mathbf{false} = \mathsf{rfalse}$ and $\sigma\mathbf{null} = \mathsf{rnull}$. We will use $r, r_1, \ldots$ to denote references, and $a, a_1, \ldots$ for fields of objects. References are atomic.

To represent the states of OO programs, we use three basic sets Name, Type and Ref. Because references in Ref are atomic, we assume two primitive functions:[1]

- $\mathsf{eqref} : \mathrm{Ref} \rightarrow \mathrm{Ref} \rightarrow \mathrm{bool}$, justifies whether two references are the same, i.e. for any $r_1, r_2 \in \mathrm{Ref}$, $\mathsf{eqref}(r_1, r_2)$ iff r_1 is same to r_2.
- $\mathit{type} : \mathrm{Ref} \rightarrow \mathrm{Type}$ decides the type of the object referred by some reference. We define $\mathit{type}(\mathsf{rtrue}) = \mathit{type}(\mathsf{rfalse}) = \mathbf{Bool}$, and $\mathit{type}(\mathsf{rnull}) = \mathbf{Null}$.

A program state $s = (\sigma, O) \in \mathrm{State}$ consists of a store and a heap. An element of O is a pair (r, f), where f is an abstraction of some object o pointed by r, a function from fields of o to values.[2] For domain of O, we refer to either a subset of Ref associated with objects, or a subset of $\mathrm{Ref} \times \mathrm{Name}$ associated with values. We use $\mathsf{dom}\, O$ to denote the domain of O, and define $\mathsf{dom}_2\, O \mathrel{\widehat{=}} \{(r, a) \mid r \in \mathsf{dom}\, O, a \in \mathsf{dom}\, O(r)\}$ for the second case.

For the program states, we define the well-typedness as follows.

Definition 3 (Well-Typed States). *State $s = (\sigma, O)$ is well-typed if both its store σ and heap O are well-typed, where store σ is well-typed if $\forall v \in \mathsf{dom}\, \sigma \cdot \mathit{type}(\sigma(v)) <: \mathit{dtype}(v)$; and heap O is well-typed if the following two conditions hold:*

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

Clearly, a well-typed store has all its variables taking values of the valid types. On the other hand, a well-typed heap requires that: 1) all fields in O are valid according to their objects, and hold values of valid types; and 2) for a non-empty object (according to its type), only when at least one of its fields is in O, we can say the object is in O. Thus we can identify empty objects in any heap. We will only consider well-typed states in our study.

---

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

[2] Please pay attention that $\mathrm{Ref} \rightarrow_{\mathsf{fin}} \mathrm{Name} \rightarrow_{\mathsf{fin}} \mathrm{Ref}$ is very different from $\mathrm{Ref} \times \mathrm{Name} \rightarrow_{\mathsf{fin}} \mathrm{Ref}$. Informally speaking, the former is a map from references to objects, while the latter is a map from object fields to field values. So objects have no direct presentations in the latter, especially empty objects.

$$\text{[I-FALSE]}\ \mathcal{M}_{\mathcal{J}}(\mathbf{false}) = \emptyset \quad \text{[I-TRUE]}\ \mathcal{M}_{\mathcal{J}}(\mathbf{true}) = \mathrm{State}$$

$$\text{[I-LOOKUP]}\ \mathcal{M}_{\mathcal{J}}(v = r) = \{(\sigma, O) \mid \sigma(v) = r\} \quad \text{[I-REF-EQ]}\ \mathcal{M}(r_1 = r_2) = \begin{cases} \mathrm{State} & \text{if } \mathsf{eqref}(r_1, r_2) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{[I-REF-TP]}\ \mathcal{M}(r : T) = \begin{cases} \mathrm{State} & \text{if } \mathit{type}(r) = T \\ \emptyset & \text{otherwise} \end{cases} \quad \text{[I-REF-STP]}\ \mathcal{M}(r <: T) = \begin{cases} \mathrm{State} & \text{if } \mathit{type}(r) <: T \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{[I-EMPTY]}\ \mathcal{M}_{\mathcal{J}}(\mathbf{emp}) = \{(\sigma, \emptyset)\} \quad \text{[I-SINGLE]}\ \mathcal{M}_{\mathcal{J}}(r_1.a \mapsto r_2) = \{(\sigma, \{(r_1, a, r_2)\})\}$$

$$\text{[I-OBJ]}\ \mathcal{M}_{\mathcal{J}}(\mathsf{obj}(r, T)) = \{(\sigma, O) \mid \mathsf{dom}\, O = \{r\} \wedge \mathsf{dom}\,(O(r)) = \mathsf{fields}(\mathit{type}(r))\}$$

$$\text{[I-APP]}\ \mathcal{M}_{\mathcal{J}}(p(\overline{r})) = \mathcal{J}(p)(\overline{r}) \quad \text{[I-EX]}\ \mathcal{M}_{\mathcal{J}}(\exists r \cdot \varphi) = \{(\sigma, O) \mid \exists r \in \mathrm{Ref} \cdot (\sigma, O) \in \mathcal{M}_{\mathcal{J}}(\varphi)\}$$

$$\text{[I-NEG]}\ \mathcal{M}_{\mathcal{J}}(\neg\varphi) = \mathrm{State} \setminus \mathcal{M}_{\mathcal{J}}(\varphi) \quad \text{[I-OR]}\ \mathcal{M}_{\mathcal{J}}(\varphi_1 \vee \varphi_2) = \mathcal{M}_{\mathcal{J}}(\varphi_1) \cup \mathcal{M}_{\mathcal{J}}(\varphi_2)$$

$$\text{[I-S-CONJ]}\ \mathcal{M}_{\mathcal{J}}(\varphi_1 * \varphi_2) = \{(\sigma, O) \mid \exists O_1, O_2 \cdot O_1 * O_2 = O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{J}}(\varphi_1) \wedge (\sigma, O_2) \in \mathcal{M}_{\mathcal{J}}(\varphi_2)\}$$

$$\text{[I-S-IMPLY]}\ \mathcal{M}_{\mathcal{J}}(\varphi_1 \mathrel{-\!\!*} \varphi_2) = \{(\sigma, O) \mid \forall O_1 \cdot O_1 \perp O \wedge (\sigma, O_1) \in \mathcal{M}_{\mathcal{J}}(\varphi_1) \text{ implies } (\sigma, O_1 * O) \in \mathcal{M}_{\mathcal{J}}(\varphi_2)\}$$

Fig. 10. Semantic for OOSL wrt. the least fixed point model $\mathcal{J}$ of the given logic environment

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

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

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

We use $O_1 \perp O_2$ to indicate that O_1 and O_2 are separated from each other:

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

If a reference, to some object o, is in both domains of two heaps O_1 and O_2, then each of O_1 and O_2 must contain a non-empty subset o's fields (the well-typedness guarantees this), and the two subsets must be disjoint. This means that we can separate fields of a non-empty object into different heaps, but not an empty object. With this definition, we have $O \perp \emptyset$ for any O, as well as $\emptyset \perp \emptyset$. When $O_1 \perp O_2$, we will use $O_1 * O_2$ for the union $O_1 \cup O_2$.

The storage model defined above, with the definition for the separation concept, gives us both an object view and a field view for the heaps. With this model, we can correctly handle both the whole objects and the fields of objects.

To define the semantics for OOSL, we need to have a careful treatment about the *user-defined assertions*. Because we allow recursive definitions (either self-recursion or mutual recursion for predicates), any reasonable definition for their semantics must involve some fixed point. We record all the definitions in a *Logic Environment* Λ:

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

Here ε denotes the empty environment, and Λ is a sequence of definitions.

As the well-formedness, the body assertion ψ of any definition in Λ cannot use symbols which is not defined in Λ. Further, we require that Λ is *finite* (of course) and any body assertion

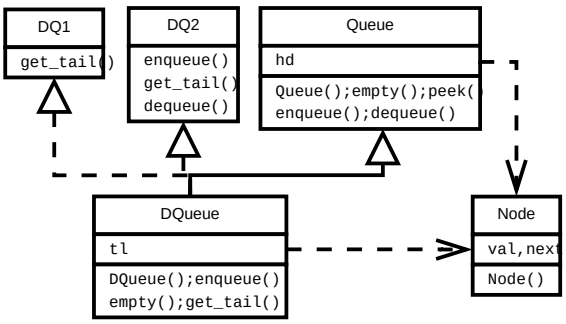


Fig. 11. the class diagram of $DQ1$, $DQ2$, $Queue$, $DQueue$, $Node$

ψ in Λ is *syntactically monotone*[3]. Under these conditions, the *least fix-point model* for a given Λ exists by Tarski's theorem. We name the least fix-point model as $\mathcal{J}$. We define the semantics of assertions as a function $\mathcal{M}_\Lambda : \Psi \to \mathbb{P}(\mathrm{State})$ based on this model by the rules listed in **Fig. 10**.

Having the semantic function, we define that an assertion ψ holds on a given state (σ, O), written $(\sigma, O) \models \varphi$, as:

$$(\sigma, O) \models \varphi \qquad \text{iff} \qquad (\sigma, O) \in \mathcal{M}(\varphi).$$

B More Examples

In this appendix, we give some more examples to illustrate how our verification framework can deal with some typical OO programming idioms with interfaces. The examples show also how we give specifications for the methods in interfaces and classes, and how the specification predicates connect the abstract specifications with the implementations.

The first example illustrates the case where a subclass implements two different interfaces, where some implementing methods are inherited from its superclass. The second example is a variant of the first one. The third example shows how to specify and verify two totally different implementations of one common interface, where we have a set interface with a linked-list and a binary-tree implementations. We also give some codes to show how our framework suppose the verification of client code completely independent from the implementation details.

B.1 Specifying and Verifying a Queue Example

In this subsection, we illustrate the verification of an example related to *queue* by using the specification and verification techniques supported by our framework.

In this example, we declare two interfaces $DQ1, DQ2$ and two classes $Queue, DQueue$ which are different implementations of FIFO *queue*s. The structural relation of the interfaces and classes in this example is visualized in a UML class diagram in **Fig. 11**.

We give part of the code in **Fig. 12**, where we define a class $Node$ where each $Node$ object (node) holds an integer and a nxt reference. We pack its structure using a specification predicate $node(\mathbf{this}, v, n)$, which asserts that a $Node$ object holds v and n as its value and the next reference. Note that this predicate is $\mathbf{pub}$, that means its definition is available everywhere. $Node$ has only a constructor with its specification.

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

```
class Node{
  def pub node(this, v, n) :
    this.val ↦ v * this.nxt ↦ n;
  Int val; Node nxt;
  Node(Int c) ⟨emp⟩⟨node(this, c, rnull)⟩
  { this.val = c; this.nxt = null; }
}
inter DQ1{
  def queue(this, α);
  Int get_tail() ⟨queue(this, β :: [a])⟩
    ⟨res = a ∧ queue(this, β :: [a])⟩;
}
inter DQ2{
  def queue(this, α);
  void enqueue(Int c)
    ⟨queue(this, α)⟩⟨queue(this, α :: [c])⟩;
  Int dequeue() ⟨queue(this, [c] :: β)⟩
    ⟨res = c ∧ queue(this, β) * true⟩;
  Int get_tail() ⟨queue(this, β :: [a])⟩
    ⟨res = a ∧ queue(this, β :: [a])⟩;
}
```

Fig. 12. Codes and specifications of class *Node* and interfaces *DQ1*, *DQ2*

Two interfaces *DQ1* and *DQ2* are declared in **Fig. 12**, where both declare a non-public specification predicate *queue* which does not have a body. Interface *DQ1* declares a method *get_tail* which returns the value of the last element in the queue. Interface *DQ2* declares two more methods *dequeue*, *enqueue* which are common for queue data structures. For specifying these methods, we use some mathematical notations. Here $[\ldots]$ denotes the sequence, and :: is the concatenation operation. We omit the **return** in *enqueue* and write its return type as **void** for convenience. All the methods in two interfaces are specified formally using abstract specifications, where *queue* is used to as the abstraction facilities.

Clearly, each implementing class of these interfaces must not only provide definitions for the methods, but also a body for *queue* to connect the specifications here with the internal implementation of the class. A client of these interfaces can see only the abstract specifications.

Class *Queue* (in the left side of **Fig. 13**) defines a kind of simple queues, 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. We encapsulate the implementing detail in the body of predicate *queue*. This predicate is also used in specifications of method definitions for *empty*, *enqueue*,*dequeue*, *peek* in *Queue*. In detail, the body of $queue(\mathbf{this}, \alpha)$ states that field *hd* of a *Queue* object refers to a single linked list recording a value sequence $[0] :: \alpha$ (Here we assume that 0 is the value of the head node.) Here we have another predicate $listseg(\mathbf{this}, r_1, r_2, \alpha)$, which is given in the common Separation Logic style. It specifies a single linked list segment between r_1 and r_2, which holds a value sequence α. This predicate is used in *queue*. Note that in the definition for *listseg*, **this** acts as a dumb parameter which connects also predicate *listseg* with the class. For the more, method *empty* returns a boolean result of whether the queue is empty, and *peek* returns the first value in the queue, while other methods are similar with the corresponding ones in interface *DQ2*. However, *Queue* has nothing to do with the two interfaces declared above.

Another class *DQueue* is defined in **Fig. 13** too. It inherits *Queue* and implements both the interfaces *DQ1* and *DQ2*. *DQueue* defines a kind of "faster queues". In it a new field *tl* is introduced which points to the last node of its list. The implementation details are hidden in the local definition of *queue*. This predicate implements the same predicate declared in both *DQ1* and *DQ2*, and overrides the predicate definition in class *Queue*. Here *DQueue* overrides method *empty* and inherits the method *peek* from its superclass *Queue*. To implement the interface, *DQueue* must implement all their declared methods. It defines *get_tail* to implement the declared method *get_tail* in *DQ1* and *DQ2*; and defines *enqueue* to implement the one in *DQ2* that also overrides the one in *Queue*. Interestingly, *DQueue* inherits *dequeue* from its superclass *Queue* with its specification to implement the one declared in *DQ2*.

```
class Queue{
  def queue(this, α) : ∃r · this.hd ↦ r∗
    listseg(this, r, rnull, [0] :: α);
  def listseg(this, r1, r2, α) :
    (α = [] ∧ r1 = r2 ∧ emp) ∨
    (∃r, b, β · (α = [b] :: β)∧
     node(r1, b, r) ∗ listseg(this, r, r2, β));
  Node hd;
  Queue() ⟨emp⟩⟨queue(this, [])⟩
  { this.hd = new Node(0); }
  Bool empty() ⟨queue(this, α)⟩
  ⟨res = (α = []) ∧ queue(this, α)⟩
  { Node p; Bool b;
    p = this.hd; p = p.nxt;
    if (p==null) b = true;
    else b = false;
    return b;
  }
  void enqueue(Int c) ⟨queue(this, α)⟩
  ⟨queue(this, α :: [c])⟩
  { Node p, q, t; p = this.hd; q = p.nxt;
    while (q!=null){ p = q; q = q.nxt; }
    t = new Node(c); p.nxt = t;
  }
  Int dequeue() ⟨queue(this, [a] :: β)⟩
  ⟨res = a ∧ queue(this, β) ∗ true⟩
  { Int c = 0; Node p, q;
    p = this.hd; q = p.nxt;
    if (q!=null)
      { c = q.val; q = q.nxt; p.nxt = q; }
    return c;
  }
  Int peek() ⟨queue(this, [c] :: β)⟩
  ⟨res = c ∧ queue(this, [c] :: β)⟩
  { Int c = 0; Node p, q;
    p = this.hd; q = p.nxt;
    if (q!=null) c = q.val;
    return c;
  }
}
class DQueue : Queue ▷ DQ1, DQ2{
  def queue(this, α) :
    ∃r, r′, b, β · ([0] :: α = β :: [b])∧
      (this.hd ↦ r ∗ this.tl ↦ r′∗
       listseg(this, r, r′, β) ∗ node(r′, b, rnull));
  Node tl;
  DQueue() ⟨emp⟩⟨queue(this, [])⟩
  { Node t = new Node(0);
    this.hd = this.tl = t;
  }
  Bool empty()
  { Node p, q; Bool b;
    p = this.hd; q = this.tl;
    if (p==q) b = true;
    else b = false;
    return b;
  }
  Int get_tail()
  { Int c = tl.val; return c; }
  void enqueue(Int c)
  { Node p, t; p = this.tl;
    t = new Node(c); p.nxt = t; this.tl = t;
  }
}
```

Fig. 13. Codes and specifications of class $Queue$, $DQueue$

Some methods in $DQueue$ have no specifications. By our rules they inherit specifications from all the supertypes of $DQueue$. Here $empty$ and $peek$ have their specifications from $Queue$ respectively. But $enqueue$ will inherit two specifications from $DQ2$ and $Queue$ similarly, and get_tail inherits two specifications from $DQ1.get\_tail$ and $DQ2.get\_tail$, respectively.

Now we turn to verify the code according to the definition of "Correct Program" using related inference rules in our verification framework before give them to clients. Firstly, we prove the simple classes $Node$ and $Queue$ are correct by rule [H-MTHD1] directly in **Fig. 14-16**.

Then, we turn to verify methods in class $DQueue$. The constructor $DQueue.DQueue$ is easy to prove by rule [H-CONSTR] (in **Fig. 17**). We can easily find that rule [H-MTHD2] should be used in verifying methods $enqueue$, get_tail, and $empty$ due to our classification of methods with the inference rules for verification. For these methods, we should verify that their body satisfy their inherited specification(s) respectively. Because the example is simple, some inherited specifications for a method are the same, then we need to verify the body only one time. For example, $DQueue.enqueue$ inherits two specifications from $DQ2$ and $Queue$ which are the same,

Proving $Node.Node$**:**

$\{\mathsf{raw}(\mathbf{this}, Node)\}$
$\mathbf{this}.val = c;\ \ \mathbf{this}.nxt = \mathbf{null};$
$\{\mathbf{this}.val \mapsto c * \mathbf{this}.nxt \mapsto \mathsf{rnull}\}$
$\{node(\mathbf{this}, c, \mathsf{rnull})\}$

Proving $Queue.Queue$**:**

$\{\mathsf{raw}(\mathbf{this}, Queue)\}$
$Node\ x;\ x = \mathbf{new}\ Node(0);$
$\{\exists r_h \cdot x = r_h \wedge \mathsf{raw}(\mathbf{this}, Queue) * node(r_h, 0, \mathsf{rnull})\}$
$\mathbf{this}.hd = x;$
$\{\exists r_h \cdot x = r_h \wedge \mathbf{this}.hd \mapsto r_h * listseg(\mathbf{this}, r_h, \mathsf{rnull}, [0])\}$
$\{queue(\mathbf{this}, [])\}$

Fig. 14. Verifying $Node.Node$ and $Queue.Queue$

$\{p = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge queue(\mathbf{this}, \alpha)\}$
$p = \mathbf{this}.hd;$
$\{\exists r_1 \cdot p = r_1 \wedge b = \mathsf{rfalse} \wedge \mathbf{this}.hd \mapsto r_1 * listseg(\mathbf{this}, r_1, \mathsf{rnull}, [0] :: \alpha)\}$
$p = p.nxt;$
$\{\exists r_1, r_2 \cdot p = r_2 \wedge b = \mathsf{rfalse} \wedge$
$\mathbf{this}.hd \mapsto r_1 * node(r_1, 0, r_2) * listseg(\mathbf{this}, r_2, \mathsf{rnull}, \alpha)\}$
$\mathbf{if}\ (p == \mathbf{null})$
$\{\exists r_1 \cdot p = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge \mathbf{this}.hd \mapsto r_1 * node(r_1, 0, \mathsf{rnull}) * listseg(\mathbf{this}, \mathsf{rnull}, \mathsf{rnull}, [])\}$
$b = \mathbf{true};$
$\{\exists r_1 \cdot p = \mathsf{rnull} \wedge b = \mathsf{rtrue} \wedge \mathbf{this}.hd \mapsto r_1 * node(r_1, 0, \mathsf{rnull})\}$
$\{p = \mathsf{rnull} \wedge b = \mathsf{rtrue} \wedge queue(\mathbf{this}, [])\}$
$\mathbf{else}$
$\{\exists r_1, r_2 \cdot p = r_2 \wedge r_2 \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge$
$\mathbf{this}.hd \mapsto r_1 * node(r_1, 0, r_2) * listseg(\mathbf{this}, r_2, \mathsf{rnull}, \alpha)\}$
$b = \mathbf{false};$
$\{\exists r_1, r_2 \cdot p = r_2 \wedge r_2 \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge$
$\mathbf{this}.hd \mapsto r_1 * node(r_1, 0, r_2) * listseg(\mathbf{this}, r_2, \mathsf{rnull}, \alpha)\}$
$\{\exists r_2 \cdot p = r_2 \wedge r_2 \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge queue(\mathbf{this}, \alpha)\}$
$\mathbf{return}\ b;$
$\{queue(\mathbf{this}, \alpha) \wedge ((\alpha = [] \wedge \mathsf{res} = \mathsf{rtrue}) \vee (\alpha \neq [] \wedge \mathsf{res} = \mathsf{rfalse}))\}$
$\{\mathsf{res} = (\alpha = []) \wedge queue(\mathbf{this}, \alpha)\}$

Fig. 15. Verifying $Queue.empty$ with its specification

we verify that its body satisfies the specification, as shown in **Fig. 17**). $DQueue.get\_tail$ inherits also two equal specification, its body verification is given in **Fig. 18**). For $DQueue.empty$, we only need to verify its body with the inherited specification from $Queue.empty$ because it just overrides the method in the superclass $Queue$ (given in **Fig. 18**).

Different to the above, we should use rule [H-MINH] for methods $dequeue$ and $peek$ of $DQueue$ to avoid re-verifying inherited method bodies. On the other hand, to prove this kind of methods, we need to prove the refinement relation for specifications, that required in the premise of rule [H-MINH], to ensure the behavioral subtyping. We give the details as follows.

For $DQueue.dequeue$, we have to prove two refinement relations of specifications because it inherits the specification of $Queue.dequeue$. One is the relation between specifications of $DQueue.dequeue$ and $DQ2.dequeue$, that asks for proving $\Gamma, DQueue \vdash \langle P\rangle\langle Q\rangle \sqsubseteq \langle P\rangle\langle Q\rangle$. This is trivially true because the two specifications are the same.

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge t = \mathsf{rnull} \wedge \mathit{queue}(\mathbf{this}, \alpha)\}$
$p = \mathbf{this}.\mathit{hd};\ q = p.\mathit{nxt};$
$\{\exists r_p, r_q \cdot p = r_p \wedge q = r_q \wedge t = \mathsf{rnull} \wedge$
$\mathbf{this}.\mathit{hd} \mapsto r_p * \mathit{node}(r_p, 0, r_q) * \mathit{listseg}(\mathbf{this}, r_q, \mathsf{rnull}, \alpha)\}$
$\mathbf{while}\ (q\ !=\mathbf{null})\ \{$
$\{\exists r_p, r_q, a, \beta, \gamma \cdot p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge ([0] :: \alpha = \beta :: [a] :: \gamma) \wedge$
$\mathit{listseg}(\mathbf{this}, \mathbf{this}.\mathit{hd}, r_p, \beta) * \mathit{node}(r_p, a, r_q) * \mathit{listseg}(\mathbf{this}, r_q, \mathsf{rnull}, \gamma)\}$
$p = q;\ q = p.\mathit{nxt};$
$\}$
$\{\exists r_p, \beta, a \cdot p = r_p \wedge q = \mathsf{rnull} \wedge t = \mathsf{rnull} \wedge$
$([0] :: \alpha = \beta :: [a]) \wedge \mathit{listseg}(\mathbf{this}, \mathbf{this}.\mathit{hd}, r_p, \beta) * \mathit{node}(r_p, a, \mathsf{rnull})\}$
$t = \mathbf{new}\ \mathit{Node}(c);\ p.\mathit{nxt} = n;$
$\{\exists r_p, r_t, \beta, a \cdot p = r_p \wedge t = r_t \wedge ([0] :: \alpha = \beta :: [a]) \wedge$
$\mathit{listseg}(\mathbf{this}, \mathbf{this}.\mathit{hd}, r_p, \beta) * \mathit{node}(r_p, a, r_t) * \mathit{node}(r_t, c, \mathsf{rnull})\}$
$\{\mathit{listseg}(\mathbf{this}, \mathbf{this}.\mathit{hd}, \mathsf{rnull}, [0] :: \alpha :: [c])\}$
$\{\mathit{queue}(\mathbf{this}, \alpha :: [c])\}$

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge c = 0 \wedge \mathit{queue}(\mathbf{this}, [a] :: \beta)\}$
$p = \mathbf{this}.\mathit{hd};\ \ q = p.\mathit{nxt};$
$\{\exists r_p, r_q \cdot p = r_p \wedge q = r_q \wedge c = 0 \wedge$
$\mathbf{this}.\mathit{hd} \mapsto r_p * \mathit{node}(r_p, 0, r_q) * \mathit{listseg}(\mathbf{this}, r_q, \mathsf{rnull}, [a] :: \beta)\}$
$\mathbf{if}\ q\,!=\mathbf{null}$
$\{\exists r_p, r_q \cdot p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge c = 0 \wedge$
$\mathbf{this}.\mathit{hd} \mapsto r_p * \mathit{node}(r_p, 0, r_q) * \mathit{listseg}(\mathbf{this}, r_q, \mathsf{rnull}, [a] :: \beta)\}$
$\{\ c = q.\mathit{val};$
$\{\exists r_p, r_q, r \cdot p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge c = a \wedge$
$\mathbf{this}.\mathit{hd} \mapsto r_p * \mathit{node}(r_p, 0, r_q) * \mathit{node}(r_q, c, r)\mathit{listseg}(\mathbf{this}, r, \mathsf{rnull}, \beta)\}$
$q = q.\mathit{nxt};\ p.\mathit{nxt} = q;$
$\{\exists r_p, r_q, r \cdot p = r_p \wedge q = r_q \wedge c = a \wedge$
$\mathbf{this}.\mathit{hd} \mapsto r_p * \mathit{node}(r_p, 0, r_q) * \mathit{node}(r, c, \mathsf{rnull}) * \mathit{listseg}(\mathbf{this}, r_q, \mathsf{rnull}, \beta)\}$
$\}$
$\{\exists r \cdot c = a \wedge \mathit{queue}(\mathbf{this}, \beta) * \mathit{node}(r, c, \mathsf{rnull})\}$
$\mathbf{return}\ \ c;$
$\{\mathsf{res} = a \wedge \mathit{queue}(\mathbf{this}, \beta) * \mathbf{true}\}$

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge c = 0 \wedge \mathit{queue}(\mathbf{this}, [a] :: \beta)\}$
$p = \mathbf{this}.\mathit{hd};\ \ q = p.\mathit{nxt};$
$\{\exists r_p, r_q \cdot p = r_p \wedge q = r_q \wedge c = 0 \wedge$
$\mathbf{this}.\mathit{hd} \mapsto r_p * \mathit{node}(r_p, 0, r_q) * \mathit{listseg}(\mathbf{this}, r_q, \mathsf{rnull}, [a] :: \beta)\}$
$\mathbf{if}\ q\,!=\mathbf{null}$
$\{\exists r_p, r_q \cdot p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge c = 0 \wedge$
$\mathbf{this}.\mathit{hd} \mapsto r_p * \mathit{node}(r_p, 0, r_q) * \mathit{listseg}(\mathbf{this}, r_q, \mathsf{rnull}, [a] :: \beta)\}$
$c = q.\mathit{val};$
$\{\exists r_p, r_q, r \cdot p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge c = a \wedge$
$\mathbf{this}.\mathit{hd} \mapsto r_p * \mathit{node}(r_p, 0, r_q) * \mathit{node}(r_q, c, r)\mathit{listseg}(\mathbf{this}, r, \mathsf{rnull}, \beta)\}$
$\{\exists r \cdot c = a \wedge \mathit{queue}(\mathbf{this}, [a] :: \beta)\}$
$\mathbf{return}\ \ c;$
$\{\mathsf{res} = a \wedge \mathit{queue}(\mathbf{this}, [a] :: \beta)\}$

Fig. 16. Verifying *Queue.enqueue*, *Queue.dequeue*, *Queue.peek*

Proving $DQueue.DQueue$**:**

$\{\mathsf{raw}(\mathbf{this}, DQueue)\}$
$Node\ t = \mathbf{new}\ \ Node(0);$
$\{\exists r_t \cdot t = r_t \wedge \mathsf{raw}(\mathbf{this}, DQueue) *$
$\quad node(r_t, 0, \mathsf{rnull})\}$
$\mathbf{this}.hd = \mathbf{this}.tl = t;$
$\{\exists r_t \cdot t = r_t \wedge \mathbf{this}.hd \mapsto r_t *$
$\quad \mathbf{this}.tl \mapsto r_t * node(r_t, 0, \mathsf{rnull})\}$
$\{\exists r_t \cdot t = r_t \wedge \mathbf{this}.hd \mapsto r_t *$
$\quad \mathbf{this}.tl \mapsto r_t *$
$\quad listseg(\mathbf{this}, r_t, r_t, []) *$
$\quad node(r_t, 0, \mathsf{rnull})\}$
$\{queue(\mathbf{this}, [])\}$

Proving $DQueue.enqueue$**:**

$\{p = \mathsf{rnull} \wedge t = \mathsf{rnull} \wedge queue(\mathbf{this}, \alpha)\}$
$p = \mathbf{this}.tl;\ t = \mathbf{new}\ \ Node(c);$
$\{\exists r, r_p, r_t, \beta, a \cdot p = r_p \wedge t = r_t \wedge$
$\quad ([0] :: \alpha = \beta :: [a]) \wedge node(r_t, c, \mathsf{rnull}) *$
$\quad (\mathbf{this}.hd \mapsto r * \mathbf{this}.tl \mapsto r_p *$
$\quad listseg(\mathbf{this}, r, r_p, \beta) * node(r_p, a, \mathsf{rnull}))\}$
$p.nxt = t;\ \mathbf{this}.tl = t;$
$\{\exists r, r_p, r_t, \beta, a \cdot p = r_p \wedge t = r_t \wedge$
$\quad ([0] :: \alpha = \beta :: [a]) \wedge$
$\quad (\mathbf{this}.hd \mapsto r * \mathbf{this}.tl \mapsto r_t * list(r, r_p, \beta) *$
$\quad node(r_p, a, r_t) * node(r_t, c, \mathsf{rnull}))\}$
$\{queue(\mathbf{this}, \alpha :: [c])\}$

Fig. 17. Verifying $DQueue.DQueue$ and $DQueue.enqueue$

Proving $DQueue.get\_tail$**:**

$\{queue(\mathbf{this}, \beta :: [a])\}$
$\{\exists r, r' \cdot c = 0 \wedge \mathbf{this}.hd \mapsto r * \mathbf{this}.tl \mapsto r' * listseg(\mathbf{this}, r, r', \beta) * node(r', a, \mathsf{rnull})\}$
$c = tl.val;$
$\{\exists r, r' \cdot c = a \wedge \mathbf{this}.hd \mapsto r * \mathbf{this}.tl \mapsto r' * listseg(\mathbf{this}, r, r', \beta) node(r', a, \mathsf{rnull})\}$
$\mathbf{return}\ \ c;$
$\{\mathsf{res} = a \wedge queue(\mathbf{this}, \beta :: [a])\}$

Proving $DQueue.empty$**:**

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge queue(\mathbf{this}, \alpha)\}$
$p = \mathbf{this}.hd;\ q = \mathbf{this}.tl;$
$\{\exists r_p, r_q, b, \beta \cdot p = r_p \wedge q = r_q \wedge b = \mathsf{rfalse} \wedge ([0] :: \alpha = \beta :: [b]) \wedge$
$\quad \mathbf{this}.hd \mapsto r_p * \mathbf{this}.tl \mapsto r_q * listseg(\mathbf{this}, r_p, r_q, \beta) * node(r_q, b, \mathsf{rnull})\}$
$\mathbf{if}\ (p{==}q)$
$\quad \{\exists r_p, r_q, b, \beta \cdot p = r_p \wedge q = r_q \wedge r_p = r_q \wedge b = \mathsf{rfalse} \wedge ([0] :: \alpha = \beta :: [b]) \wedge \beta = [] \wedge$
$\quad \mathbf{this}.hd \mapsto r_p * \mathbf{this}.tl \mapsto r_q * listseg(\mathbf{this}, r_p, r_q, \beta) * node(r_q, b, \mathsf{rnull})\}$
$\quad \{\exists r_p \cdot p = r_p \wedge b = \mathsf{rfalse} \wedge \mathbf{this}.hd \mapsto r_p * \mathbf{this}.tl \mapsto r_p *$
$\quad\quad listseg(\mathbf{this}, r_p, r_p, []) * node(r_p, 0, \mathsf{rnull})\}$
$\quad b = \mathbf{true};$
$\quad \{b = \mathsf{rtrue} \wedge queue(\mathbf{this}, [])\}$
$\mathbf{else}$
$\quad \{\exists r_p, r_q, b, \beta \cdot p = r_p \wedge q = r_q \wedge r_p \neq r_q \wedge b = \mathsf{rfalse} \wedge ([0] :: \alpha = \beta :: [b]) \wedge \beta \neq [] \wedge$
$\quad\quad \mathbf{this}.hd \mapsto r_p * \mathbf{this}.tl \mapsto r_q * listseg(\mathbf{this}, r_p, r_q, \beta) * node(r_q, b, \mathsf{rnull})\}$
$\quad b = \mathbf{false};$
$\{\exists r_1, r_2 \cdot p = r_2 \wedge r_2 \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge \mathbf{this}.hd \mapsto r_1 *$
$\quad\quad node(r_1, 0, r_2) * listseg(\mathbf{this}, r_2, \mathsf{rnull}, \alpha)\}$
$\{\exists r_2 \cdot p = r_2 \wedge r_2 \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge queue(\mathbf{this}, \alpha)\}$
$\mathbf{return}\ b;$
$\{queue(\mathbf{this}, \alpha) \wedge ((\alpha = [] \wedge \mathsf{res} = \mathsf{rtrue}) \vee (\alpha \neq [] \wedge \mathsf{res} = \mathsf{rfalse}))\}$
$\{\mathsf{res} = (\alpha = []) \wedge queue(\mathbf{this}, \alpha)\}$

Fig. 18. Verifying $DQueue.get\_tail$ and $DQueue.empty$

However, we need also to prove refinement relation between specifications of $Queue.dequeue$ and $DQueue.dequeue$. Here the situation is different, because it asks for the proof of:

$$\Gamma, DQueue \vdash \langle P\rangle\langle Q\rangle \sqsubseteq \langle \mathsf{fix}(Queue, P)\rangle\langle \mathsf{fix}(Queue, Q)\rangle$$

By the **Definition 1**, we need to prove that there exists an assertion R that

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

where

$$P = queue(\mathbf{this}, [a] :: \beta) \ \text{ and } \ Q = (\mathsf{res} = a \wedge queue(\mathbf{this}, \beta) * \mathbf{true})$$

By definition of fix, we have

$$\begin{array}{l} \Gamma, DQueue, dequeue \vdash (P \Rightarrow \mathsf{fix}(Queue, P) * R) \\ \Leftrightarrow (queue(\mathbf{this}, [a] : \beta) \Rightarrow Queue.queue(\mathbf{this}, [a] :: \beta) * R) \end{array}$$

and

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

So, the key point is to prove

$$\Gamma, DQueue, dequeue \vdash Queue.queue(r, \beta) * R \Leftrightarrow queue(r, \beta)$$

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

$$\begin{array}{l} \Gamma,\ DQueue, dequeue \vdash \\ \quad Queue.queue(r, \beta) * R \\ \Leftrightarrow \exists r_h \cdot r.hd \mapsto r_h * listseg(\mathbf{this}, r_h, \mathsf{rnull}, [0] :: \beta) * (\exists r_t \cdot r.tl \mapsto r_t) \\ \Leftrightarrow \exists r_h, r_t \cdot r.hd \mapsto r_h * r.tl \mapsto r_t * listseg(\mathbf{this}, r_h, \mathsf{rnull}, [0] :: \beta) \\ \Leftrightarrow queue(r, \beta) \end{array}$$

Thus, we have proven that $DQueue.dequeue$ meets its specifications.

For $DQueue.peek$, we only need to prove the refinement relation of specifications:

$$\Gamma, DQueue \vdash \langle P\rangle\langle Q\rangle \sqsubseteq \langle \mathsf{fix}(Queue, P)\rangle\langle \mathsf{fix}(Queue, Q)\rangle$$

It is similar to the above proof for $DQueue.dequeue$. We omit its details.

Until now, we have proven that classes $Node,\ Queue,\ DQueue$ with interfaces $DQ1,\ DQ2$ are correct by using our modular verification framework. Now we can use these interfaces and classes for some purpose. We show next how the client code of them can be specified and verified without touch the implementation details of these classes. In **Fig. 19** (left) we give a simple client method with specification, and use the code to show how our modularity and information hiding mechanisms work very well. The proof is give in the figure too.

In detail, here we define a method `qtest` which declares a variable $q1$ of class $Queue$ but actually initializes it by an queue object of class $DQueue$. Then a new variable $q2$ of interface $DQ2$ is declared which is made to refer to the same object by casting. By judging that $q1$ is constructed to be empty, we insert two integers $1, 2$ into the queue via $q1$. After doing these, we check whether the first element's value of the queue $q1$ refers equals to the last element's value of the queue $q2$ refers. If they are not equal, we pop one element from the queue denoted by $q1$ and one element from the queue denoted by $q2$, sequentially. By comparing whether these two popped values are equal, we get the final result of the method. The verification of the method is totally on an abstract level where no implementation details is used.

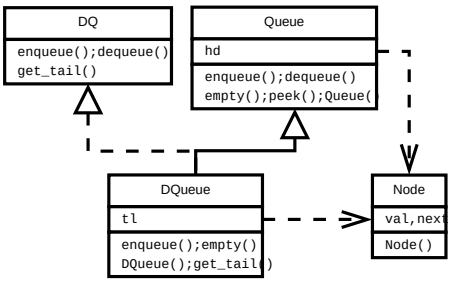
```
Bool qtest()
⟨true⟩⟨res = rtrue⟩
{
  Int i = 1; Int c1, c2;
  Bool b = false;
  Queue q1 =
    new DQueue();
  DQ2 q2 = (DQ2)q1;
  b = q1.empty();
  if (b==true){
    while (i < 3){
      q1.enqueue(i);
      i = i + 1;
    }
  }
  c1 = q1.peek();
  c2 = q2.get_tail();
  if (c1 !=c2){
    c1 = q1.dequeue();
    c2 = q2.dequeue();
  }
  if (c1==c2)
    b = false;
  return b;
}
```

```
{i = 1 ∧ c1 = 0 ∧ c2 = 0 ∧ b = rfalse ∧ q1 = rnull ∧ q2 = rnull}
i = 1; b = false; q1 = new DQueue(); q2 = (DQ2)q1;
{∃r1, r2 · i = 1 ∧ c1 = 0 ∧ c2 = 0 ∧ b = rfalse ∧ q1 = r1 ∧
          q2 = r2 ∧ r2 = r1 ∧ queue(r1, [])}
b = q1.empty();
{∃r1, r2 · i = 1 ∧ c1 = 0 ∧ c2 = 0 ∧ b = rtrue ∧ q1 = r1 ∧
          q2 = r2 ∧ r2 = r1 ∧ queue(r1, [])}
if (b==true)
  {∃r1, r2 · i = 1 ∧ c1 = 0 ∧ c2 = 0 ∧ b = rtrue ∧ q1 = r1 ∧
            q2 = r2 ∧ r2 = r1 ∧ queue(r1, [])}
{ while (i < 3){
    {∃r1, r2, β · i < 3 ∧ q1 = r1 ∧ q2 = r2 ∧ r2 = r1 ∧
              queue(r1, β)}
    q1.enqueue(i); i = i + 1;
    {∃r1, r2, β · i ≤ 3 ∧ q1 = r1 ∧ q2 = r2 ∧ r2 = r1 ∧
              queue(r1, β :: [i − 1])} }
  {∃r1, r2 · i = 3 ∧ q1 = r1 ∧ q2 = r2 ∧ r2 = r1 ∧
            queue(r1, [1] :: [2])}
}
{∃r1, r2 · c1 = 0 ∧ c2 = 0 ∧ b = rtrue ∧ i = 3 ∧ q1 = r1 ∧
          q2 = r2 ∧ r2 = r1 ∧ queue(r1, [1] :: [2])}
c1 = q1.peek(); c2 = q2.get_tail();
{∃r1, r2 · c1 = 1 ∧ c2 = 2 ∧ b = rtrue ∧ i = 3 ∧ q1 = r1 ∧
          q2 = r2 ∧ r2 = r1 ∧ queue(r1, [1] :: [2])}
if (c1 !=c2)
  {∃r1, r2 · c1 = 1 ∧ c2 = 2 ∧ c1 ≠ c2 ∧ b = rtrue ∧ i = 3 ∧
            q1 = r1 ∧ q2 = r2 ∧ r2 = r1 ∧ queue(r1, [1] :: [2])}
  { c1 = q1.dequeue(); c2 = q2.dequeue(); }
{∃r1, r2 · c1 = 1 ∧ c2 = 2 ∧ b = rtrue ∧ i = 3 ∧ q1 = r1 ∧
          q2 = r2 ∧ r2 = r1 ∧ queue(r1, [])}
if (c1==c2)
  {∃r1, r2 · c1 = 1 ∧ c2 = 2 ∧ c1 = c2 ∧ b = rtrue ∧ i = 3 ∧
            q1 = r1 ∧ q2 = r2 ∧ r2 = r1 ∧ queue(r1, [])}
  {false}
  b = false;
{∃r1, r2 · c1 = 1 ∧ c2 = 2 ∧ c1 ≠ c2 ∧ b = rtrue ∧ i = 3 ∧
          q1 = r1 ∧ q2 = r2 ∧ r2 = r1 ∧ queue(r1, [])}
return b;
{res = rtrue}
```

Fig. 19. Client Method $qtest$ with its Verification

B.2 Specifying and Verifying a Variant of the Queue Example

In above example, we use two interfaces in the interface/class hierarchy to show the application of our verification framework fully. However, the two interfaces in the example are not really necessary in practice, and they can be combined to one interface $DQ2$ which provides enough behavioral interfaces for its implementing class $DQueue$. Now we develop and specify a variant of the example in **Fig. 20**. Here we list only interface $DQ2$ the same as in **Fig. 12**, and class



$$
\begin{array}{l}
\textbf{inter}\ DQ2\{ \\
\quad \textbf{def}\ queue(\textbf{this}, \alpha); \\
\quad \textbf{void}\ enqueue(\textbf{Int}\ c);\ \langle\cdots\rangle\,\langle\cdots\rangle \\
\quad \textbf{Int}\ dequeue();\ \langle\cdots\rangle\,\langle\cdots\rangle \\
\quad \textbf{Int}\ get\_tail();\ \langle\cdots\rangle\,\langle\cdots\rangle \\
\} \\
\textbf{class}\ DQueue : Queue \rhd DQ2\{ \\
\quad \textbf{def}\ queue(\textbf{this}, \alpha) : \cdots ; \\
\quad Node\ tl; \\
\quad DQueue()\ \langle\cdots\rangle\,\langle\cdots\rangle\{\cdots\} \\
\quad \textbf{Bool}\ empty()\ \{\cdots\} \\
\quad \textbf{Int}\ get\_tail()\ \{\cdots\} \\
\quad \textbf{void}\ enqueue(\textbf{Int}\ c)\ \{\cdots\} \\
\}
\end{array}
$$

Fig. 20. Abbrev. of interface $DQ2$ and class $DQueue$ with whole UML diagram

$DQueue$ which is declared to implement $DQ2$ while remaining to inherit class $Queue$. Except these, all the other codes, specifications of class $Node$, $Queue$ are same as what in last example. Moveover, their verifications remain unchanged.

For the modification in $DQueue$, we should take into account the specifications and verifications for methods in it again. However, we find that the interface $DQ1$ in last example just declares one method get_tail. Thus, except method $DQueue.get\_tail$, other methods in $DQueue$ can be proved with their specification(s) in completely the same ways as what we have done before. For method $DQueue.get\_tail$, we should prove its body with the only inherited specification of $DQ2.get\_tail$ by rule [H-MTHD2]. Actually as the specifications of $DQ1.get\_tail$ and $DQ2.get\_tail$ are the same, we need still verify $DQueue.get\_tail$'s body once. All of these analysis tell us that, the verification of class $DQueue$ in this modified example is the same as the verification of $DQueue$ done above. Therefore, we can conclude that this modified code provides the same functionality (except no interface $DQ1$ here any more), and the client code in **Fig. 19** will still work correctly.

B.3 Specifying and Verifying a Set Example

As in Java and other OO languages, an interface can also accept multiple implementations in our specification and verification framework. In this subsection, we design an example following partially the design of interface `Set` with its implementing classes in Java standard library. We declare an interface Set and then implement it by two classes with different internal structures, where one takes the single linked list and another uses the binary tree.

The declaration of interface Set is given in **Fig. 21**, where a specification predicate set and three methods with their specifications are given. Informally, $set(p, \alpha)$ states that p refers to an object which holds elements of a set α. Here we use some standard set theory notations, e.g., $\in$, $\subseteq$, $\cup$, $\cap$ in method specifications. It is easy to know that, method $empty$ judges whether the set (currently) is empty; add adds a value to the set when it is not already there; and $contain$ judges whether the set contains the given value. These methods should be implemented with suitable techniques by the classes which implement Set. The meaning of *suitable* is that, in the implementation, these methods must satisfy their specifications given in interface Set.

```
inter Set{
  def set(this, α);
  Bool empty()⟨set(this, α)⟩
  ⟨set(this, α) ∧ res = (α = ∅)⟩;
  void add(Int c)⟨set(this, α)⟩
  ⟨set(this, {c} ∪ α)⟩;
  Bool contain(Int c)⟨set(this, α)⟩
  ⟨set(this, α) ∧ res = (c ∈ α)⟩;
}
```

Fig. 21. Codes and specifications of Interface *Set*

```
class Node{
  Int val; Node nxt;
  def pub node(this, v, r) :
    this.val ↦ v * this.nxt ↦ r;
  Node(Int v)⟨emp⟩⟨node(this, old(v), rnull)⟩
  {this.val = v; this.nxt = null;}
}
class ListSet ▷ Set {
  Node hd;
  def set(this, α) : ∃ s, r_h · g(α, s)∧
    this.hd ↦ r_h * listseg(this, r_h, rnull, [0] :: s);
  def listseg(this, r1, r2, s) :
    (s = [] ∧ r1 = r2 ∧ emp)∨
    (∃ r3, b, s' · (s = [b] :: s')∧
    node(r1, b, r3) * listseg(this, r3, r2, s'));
  def f(s) : (f([]) = ∅)∨
    ((s = [b] :: s') ∧ (f([b] :: s') = {b} ∪ f(s')));
  ListSet()⟨emp⟩⟨set(this, [])⟩;
  { Node x; x = new Node(0);
    this.hd = x;
  }
  Bool empty()
  { Node p, q; Bool b;
    p = this.hd; q = p.nxt;
    if (q == null) b = true;
    else b = false;
    return b;
  }
  Bool contain(Int c)
  { Node p, q; Bool b; b = false;
    p = this.hd; q = p.nxt;
    while (q!=null ∧ b!=true){
      if (q.val == c) b = true;
      q = q.nxt; }
    return b;
  }
  void add(Int c)
  { Node p, q, n; Bool b;
    p = this.hd; q = p.nxt;
    b = this.contain(c);
    if (b == rfalse) {
      n = new Node(c);
      n.nxt = q; p.nxt = n; }
  }
}
```

Fig. 22. Set Implemented as Linked List

B.4 Implementing *Set* by Single Linked List

Firstly, we implement interface *Set* by a class *ListSet* (**Fig. 22**), which uses internally a single linked list to hold the elements. Objects of class *Node* are used to build the linked lists, while each node holds an integer value with a *nxt* reference. The code, specification (and also verification) of *Node* is the same as in the queue example in Subsection B.1. We omit its details here.

Before going into the detail specifications and implementation of *ListSet*, we explain some noteworthy points. Firstly, element sequences of linked lists specified here are always *ordered but non-duplicated*. Secondly, in notation, we suppose that a formula like $s = s_1 :: s_2$ means sequence s can be divided into two sub-sequences s_1, s_2, each of which is non-duplicated. As a simple situation, if $s = [b] :: s_1$, then we know that element b is contained in sequence s but not in its sub-sequence s_1. In the following, we will simply call *sequence*s for our special non-duplicated sequences if no ambiguous.

In class *ListSet*, we implement all the predicates and methods declared in *Set*. The predicate *set* is embodied based on predicate *listseg* which specifies a segment of single linked list built

from $Node$. Thus, we indeed hide the specifically realizing structure of $ListSet$ under Set by the definition body of set from the unexpected clients.

Actually, the mathematical concepts used for element storing structures in predicate set and its auxiliary predicate $listseg$ are different, i.e., ordinary (unordered and non-duplicated) set α for set and our special (ordered and non-duplicated) sequence s for $listseg$. For reasoning, we introduce an additional logical function $f(s)$ to transform sequences to sets. Please note that, because we only talk about non-duplicated sequences, a set transformed from a sequence s (i.e., $f(s)$) has the same size and elements with s. And we will use assertion $g(\alpha, s)$ to assert that there exists a sequence s which is some permutation of elements in set α and has the same size and elements with α. By using these two mathematical notations, we define predicate set in $ListSet$ naturally by integrating predicate $listseg$ and asserting $g(\alpha, s)$ is true.

Moveover, now we can judge whether an element b is in a sequence s by equally judging whether $b \in f(s)$ is true. And as above suppose, notations like $s = s_1 :: s_2$, which mean divisions of a sequence s into two disjoint and unrepeatable sub-sequences s_1, s_2, are agreed to equally imply $f(s_1) \cap f(s_2) = \emptyset$ hold. However, we will just take these implicative properties in this kind of notations as our common sense and do not write them down explicitly and repeatedly in later specifications and verifications. Such as, when meet with notations as $s = s_1 :: [v] :: s_2$ in later verifications, we can directly get the following information, that is, $v \notin f(s_1) \wedge v \notin f(s_2) \wedge f(s_1) \cap f(s_2) = \emptyset$.

There are also some properties of assertion $g(\alpha, s)$ specified in **Lemma. 1** as below.

Lemma 1. *We will have assertion $g(\alpha, s)$ holds, if it is one of the forms as below:*

$$\begin{gathered}\alpha = \emptyset \wedge s = [] \Rightarrow g(\emptyset, []) \\ \alpha = \{a\} \wedge s = [a] \Rightarrow g(\{a\}, [a]) \\ \alpha = \{a\} \cup \beta \wedge s = [a] :: s' \wedge g(\beta, s') \Rightarrow g(\{a\} \cup \beta, [a] :: s')\end{gathered}$$

Class $ListSet$ implements all the declared methods in Set with inheriting their specifications from it. Here, specifications for these methods use the embodied predicates in $ListSet$ to hide actual structure and implementations of $ListSet$ from clients. Because we use an unrepeatable sequence for $ListSet$, the implementation algorithm of method add is that, a given element c would be added into a $ListSet$ object just when it is not already contained. That is, we should maintain the unrepeatability during all operations on $ListSet$ objects. Then if we have a *delete* method, we only need to delete the first element (if it is contained) we search for through the list and then return the result. The Other methods $empty$, $contain$ in $ListSet$ are trivial.

After designing implementations and specifying method specifications of class $ListSet$, we need to verify that class $ListSet$ meets its specifications, thus it is a correct implementation of Set. That is, we should verify each method of $ListSet$ by using some corresponding inference rule in our modular verification framework correctly. Actually, for $ListSet$, we only need to verify each method (except the constructor) body meeting its inherited specification from Set by rule [H-MTHD2] directly.

Before verifying $ListSet$, we firstly prove that class $Node$ is correct with respect to its specification. Because in class $Node$ there is only a constructor, we need only to prove that the constructor is correct by rule [H-CONSTR] (in the left side of **Fig. 23**).

Now we turn to verification of the methods in $ListSet$. As analyzing above, we should prove each of their bodies meeting its specification. Similar to $Node.Node$, we prove the constructor $ListSet.ListSet$ by rule [H-CONSTR] (in the right side of **Fig. 23**). As analyzing above, for other methods in $ListSet$, we will use the reference rule [H-MTHD2] to verify them as below.

Proving $Node.Node$**:**

$\{\mathsf{raw}(\mathbf{this}, Node)\}$
$\mathbf{this}.val = v;\ \ \mathbf{this}.nxt = \mathbf{null};$
$\{\mathbf{this}.val \mapsto v * \mathbf{this}.nxt \mapsto \mathsf{rnull}\}$
$\{node(\mathbf{this}, v, \mathsf{rnull})\}$

Proving $ListSet.ListSet$**:**

$\{x = \mathsf{rnull} \wedge \mathsf{raw}(\mathbf{this}, Node)\}$
$x = \mathbf{new}\ \ Node(0);$
$\{\exists\, r_h \cdot x = r_h \wedge \mathsf{raw}(\mathbf{this}, Node) * node(r_h, 0, \mathsf{rnull})\}$
$\mathbf{this}.hd = x;$
$\{\exists\, r_h \cdot x = r_h \wedge g(\emptyset, []) \wedge \mathbf{this}.hd \mapsto r_h *$
$\quad listseg(\mathbf{this}, r_h, \mathsf{rnull}, [0] :: [])\}$
$\{set(\mathbf{this}, \emptyset)\}$

Fig. 23. Proofs for $Node.Node$ and $ListSet.ListSet$

• $ListSet.empty$ **with its specification.** We do its body verification here:

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge set(\mathbf{this}, \alpha)\}$
$p = \mathbf{this}.hd;$
$\{\exists\, s, r_p, r_q \cdot p = r_p \wedge q = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge g(\alpha, s) \wedge$
$\quad \mathbf{this}.hd \mapsto r_p * listseg(\mathbf{this}, r_p, \mathsf{rnull}, [0] :: s)\}$
$q = p.nxt;$
$\{\exists\, s, r_p, r_q \cdot p = r_p \wedge q = r_q \wedge b = \mathsf{rfalse} \wedge g(\alpha, s) \wedge$
$\quad \mathbf{this}.hd \mapsto r_p * node(r_p, 0, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
$\mathbf{if}\ \ (q == \mathbf{null})$
$\quad \{\exists\, r_p \cdot p = r_p \wedge q = \mathsf{rnull} \wedge g(\emptyset, []) \wedge b = \mathsf{rfalse} \wedge$
$\quad\quad \mathbf{this}.hd \mapsto r_p * node(r_p, 0, \mathsf{rnull}) * listseg(\mathbf{this}, \mathsf{rnull}, \mathsf{rnull}, [])\}$
$\quad \{q = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge set(\mathbf{this}, \emptyset))\}$
$\quad b = \mathbf{true};$
$\quad \{q = \mathsf{rnull} \wedge b = \mathsf{rtrue} \wedge set(\mathbf{this}, \emptyset)\}$
$\mathbf{else}$
$\quad \{\exists\, s, r_p, r_q \cdot p = r_p \wedge q = r_q \wedge g(\alpha, s) \wedge r_q \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge$
$\quad\quad \mathbf{this}.hd \mapsto r_p * node(r_p, 0, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
$\quad \{\exists r_p, r_q \cdot p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge set(\mathbf{this}, \alpha)\}$
$\quad b = \mathbf{false};$
$\quad \{b = \mathsf{rfalse} \wedge set(\mathbf{this}, \alpha)\}$
$\mathbf{return}\ \ b;$
$\{set(\mathbf{this}, \alpha) \wedge ((\alpha = \emptyset \wedge \mathsf{res} = \mathsf{rtrue}) \vee (\alpha \neq \emptyset \wedge \mathsf{res} = \mathsf{rfalse}))\}$
$\{set(\mathbf{this}, \alpha) \wedge \mathsf{res} = (\alpha = \emptyset)\}$

Then we conclude that $ListSet.empty$ meets its specification.

• $ListSet.contain$ **with its specification.** We verify its body here:

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge set(\mathbf{this}, \alpha)\}$
$b = \mathbf{false};\ p = \mathbf{this}.hd;\ q = p.nxt;$
$\{\exists s, r_p, r_q \cdot p = r_p \wedge g(\alpha, s) \wedge q = r_q \wedge b = \mathsf{rfalse} \wedge$
$\quad \mathbf{this}.hd \mapsto r_p * node(r_p, 0, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
$\mathbf{while}\ (q\,!{=}\mathbf{null} \wedge b\,!{=}\mathbf{true})\{$
$\quad \{\exists s, r_1, r_p, r_q, v, s_1, s_2 \cdot (s = s_1 :: [v] :: s_2) \wedge p = r_p \wedge q = r_q \wedge g(\alpha, s) \wedge$
$\qquad ((b = \mathsf{rfalse} \wedge r_p.nxt \mapsto r_q) \vee (c \notin f(s_1) \wedge$
$\qquad ((v \neq c \wedge b = \mathsf{rfalse}) \vee (v = c \wedge b = \mathsf{rtrue})) \wedge r_q \neq \mathsf{rnull}))\wedge$
$\quad \mathbf{this}.hd \mapsto r_p * listseg(\mathbf{this}, r_p, r_q, [0] :: s_1) * node(r_q, v, r_1) * listseg(\mathbf{this}, r_1, \mathsf{rnull}, s_2)\}$
$\quad \mathbf{if}\ (q.val == c)$
$\qquad \{\exists s, r_1, r_p, r_q, v, s_1, s_2 \cdot (s = s_1 :: [v] :: s_2) \wedge g(\alpha, s) \wedge c \notin f(s_1) \wedge p = r_p \wedge$
$\qquad\quad q = r_q \wedge r_q \neq \mathsf{rnull} \wedge (v = c \wedge b = \mathsf{rtrue}) \wedge \mathbf{this}.hd \mapsto r_p *$
$\qquad\quad listseg(\mathbf{this}, r_p, r_q, [0] :: s_1) * node(r_q, v, r_1) * listseg(\mathbf{this}, r_1, \mathsf{rnull}, s_2)\}$
$\qquad b = \mathbf{true};$
$\qquad \{\exists s, r_p, v \cdot g(\alpha, s) \wedge v \in f(s) \wedge p = r_p \wedge (v = c \wedge b = \mathsf{rtrue}) \wedge$
$\qquad\quad \mathbf{this}.hd \mapsto r_p * listseg(\mathbf{this}, r_p, \mathsf{rnull}, [0] :: s)\}$
$\qquad \{\exists s, v, s_1, s_2 \cdot g(\alpha, s) \wedge v \in f(s) \wedge c = v \wedge b = \mathsf{rtrue} \wedge set(\mathbf{this}, \alpha)\}$

$\quad \{\exists s, r_1, r_p, r_q, v, s_1, s_2 \cdot (s = s_1 :: [v] :: s_2) \wedge g(\alpha, s) \wedge c \notin f(s_1) \wedge p = r_p \wedge q = r_q \wedge$
$\qquad ((v \neq c \wedge b = \mathsf{rfalse}) \vee (v = c \wedge b = \mathsf{rtrue})) \wedge r_q \neq \mathsf{rnull} \wedge \mathbf{this}.hd \mapsto r_p *$
$\qquad listseg(\mathbf{this}, r_p, r_q, [0] :: s_1) * node(r_q, v, r_1) * listseg(\mathbf{this}, r_1, \mathsf{rnull}, s_2)\}$
$\quad q = q.nxt;$
$\quad \{\exists s, r_1, r_p, r_q, v, v', s_1, s_2, s_2' \cdot (s = (s_1 :: [v]) :: [v'] :: s_2') \wedge g(\alpha, s) \wedge (s_2 = [v'] :: s_2') \wedge$
$\qquad c \notin f(s_1) \wedge p = r_p \wedge q = r_q \wedge ((v \neq c \wedge b = \mathsf{rfalse}) \vee (v = c \wedge b = \mathsf{rtrue})) \wedge$
$\qquad ((r_q \neq \mathsf{rnull} \wedge node(r_q, v', r_1) * listseg(\mathbf{this}, r_1, \mathsf{rnull}, s_2')) \vee (r_q = \mathsf{rnull})) *$
$\qquad \mathbf{this}.hd \mapsto r_p * listseg(\mathbf{this}, r_p, r_q, [0] :: s_1 :: [v])\}$
$\}$
$\{\exists s, r_q, r_p, v, s_1, s_2 \cdot ((c \notin f(s) \wedge r_q = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge g(\alpha, s) \wedge listseg(\mathbf{this}, r_p, \mathsf{rnull}, [0] :: s)) \vee$
$\quad (p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge (s = s_1 :: [v] :: s_2) \wedge v = c \wedge b = \mathsf{rtrue} \wedge$
$\quad listseg(\mathbf{this}, r_p, r_q, [0] :: s_1) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, [v] :: s_2))) \wedge\ \mathbf{this}.hd \mapsto r_p\}$
$\{((c \notin \alpha \wedge b = \mathsf{rfalse}) \vee (c \in \alpha \wedge b = \mathsf{rtrue})) \wedge set(\mathbf{this}, \alpha)\}$
$\mathbf{return}\ b;$
$\{set(\mathbf{this}, \alpha) \wedge ((c \notin \alpha \wedge \mathsf{res} = \mathsf{rfalse}) \vee (c \in \alpha \wedge \mathsf{res} = \mathsf{rtrue}))\}$
$\{set(\mathbf{this}, \alpha) \wedge \mathsf{res} = (c \in \alpha)\}$

Then we conclude that *contain* meets its specification.

• $ListSet.add$ **with its specification.** Similarly for its body:

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge n = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge set(\mathbf{this}, \alpha)\}$
$p = \mathbf{this}.hd;$
$\{\exists\, s, r_p \cdot p = r_p \wedge g(\alpha, s) \wedge q = \mathsf{rnull} \wedge n = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge \mathbf{this}.hd \mapsto r_p *$
$\quad listseg(\mathbf{this}, r_p, \mathsf{rnull}, [0] :: s)\}$
$q = p.nxt;$
$\{\exists\, s, r_p, r_q \cdot p = r_p \wedge g(\alpha, s) \wedge q = r_q \wedge n = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge \mathbf{this}.hd \mapsto r_p *$
$\quad node(r_p, 0, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
$b = \mathbf{this}.contain(c);$
$\{\exists\, s, r_p, r_q \cdot p = r_p \wedge g(\alpha, s) \wedge q = r_q \wedge n = \mathsf{rnull} \wedge ((c \notin \alpha \wedge b = \mathsf{rfalse}) \vee (c \in \alpha \wedge b = \mathsf{rtrue})) \wedge$
$\quad \mathbf{this}.hd \mapsto r_p * node(r_p, 0, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
if $(b == \mathsf{rfalse})$
$\quad \{\exists\, s, r_p, r_q \cdot p = r_p \wedge g(\alpha, s) \wedge q = r_q \wedge n = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge c \notin \alpha \wedge \mathbf{this}.hd \mapsto r_p *$
$\quad\quad node(r_p, 0, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
{
$\quad n = \mathbf{new}\ Node(c);$
$\quad \{\exists\, s, r_n, r_p, r_q \cdot p = r_p \wedge g(\alpha, s) \wedge q = r_q \wedge b = \mathsf{rfalse} \wedge c \notin \alpha \wedge n = r_n \wedge node(r_n, c, \mathsf{rnull}) *$
$\quad\quad \mathbf{this}.hd \mapsto r_p * node(r_p, 0, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
$\quad n.nxt = q;$
$\quad \{\exists\, s, r_n, r_p, r_q \cdot p = r_p \wedge g(\alpha, s) \wedge q = r_q \wedge n = r_n \wedge b = \mathsf{rfalse} \wedge c \notin \alpha \wedge node(r_n, c, r_q) *$
$\quad\quad \mathbf{this}.hd \mapsto r_p * node(r_p, 0, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
$\quad p.nxt = n;$
$\quad \{\exists\, s, r_n, r_p, r_q \cdot p = r_p \wedge g(\alpha, s) \wedge q = r_q \wedge n = r_n \wedge b = \mathsf{rfalse} \wedge c \notin \alpha \wedge \mathbf{this}.hd \mapsto r_p *$
$\quad\quad node(r_p, 0, r_n) * node(r_n, c, r_q) * listseg(\mathbf{this}, r_q, \mathsf{rnull}, s)\}$
$\quad \{\exists\, s, r_p \cdot\ c \notin \alpha \wedge g(\alpha, s) \wedge \mathbf{this}.hd \mapsto r_p * listseg(\mathbf{this}, r_p, \mathsf{rnull}, [0] :: [c] :: s)\}$
}
$\{c \in \alpha \wedge set(\mathbf{this}, \alpha)) \vee (c \notin \alpha \wedge set(\mathbf{this}, \{c\} \cup \alpha))\}$
$\{c \in \alpha \wedge set(\mathbf{this}, \{c\} \cup \alpha)) \vee (c \notin \alpha \wedge set(\mathbf{this}, \{c\} \cup \alpha))\}$
$\{set(\mathbf{this}, \{c\} \cup \alpha)\}$

Then we conclude that add meets its specification.

Until now, we have proven that the constructor of $ListSet$ meets its specification and all methods ($empty$, $contain$, add) meet their method specifications which are inherited respectively from Set. Thus, we have verified the correctness of class $ListSet$ and conclude that it is a correct implementation of interface Set finally.

B.5 Implementing Set by Binary Search Tree

We all know that implementing sets by linked lists is not efficient enough for many applications, especially because the $contain$ method requires linear time to give the result. Now, we consider to implement Set with the binary search tree, and define class $TreeSet$ for this as shown in **Fig. 24**.

Nodes of the $TreeSet$ are of the type $BNode$, which holds an integer value and two references to its left and right children. We define a public specification predicate $bnode$ to encapsulate this internal information of $BNode$ objects. In each $TreeSet$ object, field rt refers to the root of the binary tree, in which the set's elements store.

Similar to $listseg$ in class $ListSet$, here we use an auxiliary predicate $subtree$ to define the specification predicate set declared in Set. Recursively defined $subtree$ hides the implementation details of binary trees. However, due to the complexity of binary tree structure, we need some additional specification predicates, i.e., $upper$, $lower$, $tpath$. In detail, $upper(\alpha, b)$ asserts that

```
class BNode{
  Int val; BNode left, right;
  def pub bnode(this, r1, v, r2) :
    this.val ↦ v * this.left ↦ r1 *
    this.right ↦ r2;
  BNode(Int v)⟨emp⟩
  ⟨bnode(this, rnull, v, rnull)⟩;
  {this.val = v; this.left = null;
    this.right = null; }
}
class TreeSet ▷ Set {
  BNode rt;
  def set(this, α) :
    ∃r · this.rt ↦ r * subtree(r, α);
  def subtree(r, α) :
    (α = ∅ ∧ r = rnull ∧ emp)∨
    (∃ r1, r2, b, β1, β2 · (α = β1 ∪ {b} ∪ β2)∧
    upper(β1, b) ∧ lower(β2, b)∧
    bnode(r, r1, b, r2) * subtree(r1, β1)
    *subtree(r2, β2));
  def upper(α, b) : ∀ x · (x ∈ α) ∧ (x < b);
  def lower(α, b) : ∀ x · (x ∈ α) ∧ (x > b);
  def tpath(p, q, c, β, γ) : (c ∉ β ∪ γ)∧
    ((p = q ∧ β = ∅ ∧ emp) ∨ (∃l, r, v, β1, β2·
    (β = β1 ∪ {v} ∪ β2) ∧ bnode(p, l, v, r)*
    ((upper(β1 ∪ γ ∪ {c}, v) ∧ lower(β2, v)∧
    tpath(l, q, c, β1, γ) * subtree(r, β2))∨
    (upper(β1, v) ∧ lower(β2 ∪ γ ∪ {c}, v)∧
    tpath(r, q, c, β2, γ) * subtree(l, β1))))));
  TreeSet()⟨emp⟩⟨set(this, ∅)⟩
  {this.rt = null; }
  Bool empty(){
    Bool b; BNode p;
    p = this.rt;
    if (p == null) b = true;
    else b = false; return b; }
  pub void add(Int c){
    BNode p, q, t; Int v; v = 0;
    p = this.rt; q = null;
    while (p!=null){
      q = p; v = q.val;
      if (v == c) p = null;
      if (v < c) p = q.right;
      else p = q.left;
    }
    if (v!=c) {
      t = new BNode(c);
      if (q == null) this.rt = t;
      else if (v < c) q.right = t;
      else if (v > c) q.left = t;
    }
  }
  Bool contain(Int c){
    BNode p, q; Bool b; Int v;
    v = 0; p = this.rt; q = null;
    b = false;
    while (p!=null ∧ b!=true){
      q = p; v = q.val;
      if (v == c) b = true;
      if (v < c) p = q.right;
      else p = q.left; }
    return b;
  }
}
```

Fig. 24. *Set* Implemented as Binary Search Tree

integer value b is larger than all elements in integer set α, and $lower(\alpha, b)$ asserts that b is smaller than all elements in α. On the other hand, $tpath$ recombines several parts which are partitioned from an original binary tree into a binary tree. More explanations about predicate $tpath$ will be talked later. Base on these predicates, predicate $subtree$ states that, either the binary tree is empty, or it contains a root r with integer value b, its left subtree with a value set β_1 and right subtree with β_2 while b is an upper bound of β_1 but a lower bound of β_2.

However, in verification of methods in $TreeSet$, the situation we have to deal with are more complicate. Exactly, we also need to reflect properties between already defined specification predicates in $TreeSet$ which are required in verifications. Thus, we specify these properties in **Lemma 2, 3**. We will explain and prove these lemmas later in detail.

Firstly, **Lemma 2** says if an element v is both an upper bound of a set β_1 and a lower bound of another set β_2, then we can know that sets β_1 and β_2 are disjoint. And if an element v is either

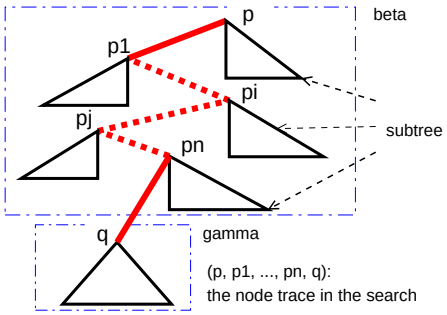


Fig. 25. searching a value in a binary tree and recording the search trace

an upper bound or a lower bound of a set, then v is not contained in this set. These properties are valid trivially in the normal set theory and we just omit their proofs here.

Lemma 2.

$$upper(\beta_1, v) \wedge lower(\beta_2, v) \Rightarrow \beta_1 \cap \beta_2 = \emptyset$$
$$upper(\beta_1, v) \Rightarrow v \notin \beta_1$$
$$lower(\beta_2, v) \Rightarrow v \notin \beta_2$$

□

Then, **Lemma 3** is more complicate than **Lemma 2**, which indicates the importance of predicate $tpath$ and deals with the difficulty we would meet with during verifying program codes in $TreeSet$.

Informally, as illustrated in **Fig. 25**, predicate $tpath(p, q, c, \beta, \gamma)$ asserts that for searching the value c from the current root p in the binary tree, we get a path from p to q and an already searched part $tpath(p, q, c, \beta, \gamma)$ while $treeset(q, \gamma)$ is the rest part to search at some time. The set β contains all values of searched nodes from the current searching root p to q's parent node p_n in the searching trace and their sub-trees' value sets. q is the node where searching of value c arrives at currently, and γ is the value set of node q and its sub-trees' value sets. Generally, we abstract a searched part of a binary tree into a predicate $tpath$ as specified in **Fig. 24** by using only arithmetics of set and sequence, and basic properties of binary tree, rather than any auxiliary tricks.

As the above describing of predicate $tpath$, suppose we fail to search c at q in the original binary tree, then we get two parts $tpath(p, q, c, \beta, \gamma)$, $subtree(q, \gamma)$ and add a new node with value c as one of q's children. Later, we should combine the new subtree $subtree(q, \gamma \cup \{c\})$ with the other part $tpath(p, q, c, \beta, \gamma)$ to form a binary tree which we specify as $subtree(p, \beta \cup \gamma \cup \{c\})$. However, it is indeed a difficult problem. Finally, we specify the right property between $treeset$ and $tpath$ in **Lemma 3** to help solving this difficulty and finishing our prove efficiently.

According to the definition of predicate $tpath$ in **Fig. 24** and the analyses above, we specify and prove its property in **Lemma 3** as following.

Lemma 3. *Suppose add any value c into a binary tree whose root is p, and the search fails at node q while the original tree is divided into two parts $tpath(p, q, c, \beta, \gamma)$ and $treeset(q, \gamma)$ as illustrated in **Fig. 25**. Value sets β and γ are described as above. If c is not found, we add a new node with value c as one of q's children and get a new sub-tree $treeset(q, \gamma \cup \{c\})$. Then we have,*

$$\beta \cap \gamma = \emptyset \Rightarrow (tpath(p, q, c, \beta, \gamma) * treeset(q, \gamma \cup \{c\}) \Rightarrow treeset(p, \beta \cup \gamma \cup \{c\}))$$

□

Proof. For the cases when $p = q$, we have:

$$\begin{aligned}
&(\beta \cap \gamma = \emptyset) \wedge tpath(p, q, c, \beta, \gamma) * subtree(q, \gamma \cup \{c\}) \\
&= (\beta \cap \gamma = \emptyset) \wedge ((c \notin \beta \cup \gamma) \wedge p = q \wedge \beta = \emptyset \wedge \mathbf{emp}) * subtree(q, \gamma \cup \{c\}) \\
&= \mathbf{true} \wedge p = q \wedge c \notin \gamma \wedge \beta = \emptyset \wedge subtree(q, \gamma \cup \{c\}) \\
&= subtree(p, \beta \cup \gamma \cup \{c\})
\end{aligned}$$

Now we suppose that $p \neq q$, and do the induction on the length k of the path $(p, p_1, \cdots, p_n, q)$ and let $k = n + 1$, which accesses from node p to q:

Base: The length of the path is $k = 1$. In this case the path is just (p, q), then

$$\begin{aligned}
&(\beta \cap \gamma = \emptyset) \wedge tpath(p, q, c, \beta, \gamma) * subtree(q, \gamma \cup \{c\}) \\
&= \exists\, l, r, v, \beta_1, \beta_2 \cdot (\beta \cap \gamma = \emptyset) \wedge (c \notin \beta \cup \gamma) \wedge (\beta = \beta_1 \cup \{v\} \cup \beta_2) \wedge bnode(p, l, v, r) * \\
&\quad ((upper(\beta_1 \cup \gamma \cup \{c\}, v) \wedge lower(\beta_2, v) \wedge l = q \wedge tpath(l, q, c, \beta_1, \gamma) * subtree(r, \beta_2)) \vee \\
&\quad (upper(\beta_1, v) \wedge lower(\beta_2 \cup \gamma \cup \{c\}, v) \wedge r = q \wedge tpath(r, q, c, \beta_2, \gamma) * subtree(l, \beta_1))) * \\
&\quad subtree(q, \gamma \cup \{c\})
\end{aligned}$$

Here we have two sub-cases:

$$\begin{aligned}
&\text{case } v > c: \\
&= \exists r, v, \beta_1, \beta_2 \cdot (\beta \cap \gamma = \emptyset) \wedge (c \notin \beta \cup \gamma) \wedge (\beta = \beta_1 \cup \{v\} \cup \beta_2) \wedge (\beta_1 = \emptyset) \wedge \\
&\quad upper(\beta_1 \cup \gamma \cup \{c\}, v) \wedge lower(\beta_2, v) \wedge bnode(p, q, v, r) * subtree(r, \beta_2) * \\
&\quad tpath(q, q, c, -, \gamma) * subtree(q, \gamma \cup \{c\}) \\
&= \exists r, v, \beta_2 \cdot (\beta \cap \gamma = \emptyset) \wedge (c \notin \beta \cup \gamma) \wedge (\beta = \{v\} \cup \beta_2) \wedge upper(\gamma \cup \{c\}, v) \wedge \\
&\quad lower(\beta_2, v) \wedge bnode(p, q, v, r) * subtree(r, \beta_2) * subtree(q, \gamma \cup \{c\}) * \mathbf{emp} \\
&= subtree(p, \beta \cup \gamma \cup \{c\})
\end{aligned}$$

$$\begin{aligned}
&\text{case } v < c: \\
&= \exists r, v, \beta_1, \beta_2 \cdot (\beta \cap \gamma = \emptyset) \wedge (c \notin \beta \cup \gamma) \wedge (\beta = \beta_1 \cup \{v\} \cup \beta_2) \wedge (\beta_2 = \emptyset) \wedge \\
&\quad upper(\beta_1, v) \wedge lower(\beta_2 \cup \gamma \cup \{c\}, v) \wedge bnode(p, l, v, q) * subtree(l, \beta_1) * \\
&\quad tpath(q, q, c, -, \gamma) * subtree(q, \gamma \cup \{c\}) \\
&= \exists r, v, \beta_2 \cdot (\beta \cap \gamma = \emptyset) \wedge (c \notin \beta \cup \gamma) \wedge (\beta = \{v\} \cup \beta_1) \wedge upper(\beta_1, v) \wedge \\
&\quad lower(\gamma \cup \{c\}, v) \wedge bnode(p, l, v, q) * subtree(l, \beta_1) * subtree(q, \gamma \cup \{c\}) * \mathbf{emp} \\
&= subtree(p, \beta \cup \gamma \cup \{c\})
\end{aligned}$$

Induction: Suppose we have proven that if the length of the path is $k \leq n$: i.e., $(p, p_1, \cdots, p_k, q)$, $k = 1, 2, \cdots, n-1$, the conclusion holds. For the case when the length of the path is $k = n+1$, i.e., $(p, p_1, \cdots, p_{n-1}, p_n, q)$, we have,

$$\begin{aligned}
&(\gamma' = \gamma \cup \beta_n) \wedge (\beta = \beta' \cup \beta_n) \\
&(\beta' \cap \gamma' = \emptyset) \wedge tpath(p, p_n, c, \beta', \gamma') * subtree(p_n, \gamma' \cup \{c\}) \Rightarrow subtree(p, \beta' \cup \gamma' \cup \{c\}) \\
&(\beta \cap \gamma = \emptyset) \wedge (\gamma' = \gamma \cup \beta_n) \wedge (\beta = \beta' \cup \beta_n) \Rightarrow (\beta' \cap \gamma' = \emptyset) \\
&(\beta \cap \gamma = \emptyset) \wedge (\beta = \beta' \cup \beta_n) \Rightarrow (\beta_n \cap \gamma = \emptyset)
\end{aligned}$$

Thus,

$$
\begin{array}{ll}
 & (\beta \cap \gamma = \emptyset) \wedge tpath(p, q, c, \beta, \gamma) * subtree(q, \gamma \cup \{c\}) \\
\Rightarrow & (\beta \cap \gamma = \emptyset) \wedge tpath(p, p_n, c, \beta', \gamma') * tpath(p_n, q, c, \beta_n, \gamma) * subtree(q, \gamma \cup \{c\}) \\
= & (\beta \cap \gamma = \emptyset) \wedge ((\beta_n = \{v_n\} \cup \beta_1 \cup \beta_2) \wedge upper(\beta_1, v_n) \wedge lower(\beta_2, v_n) \wedge \\
 & \quad tpath(p, q_n, c, \beta', \gamma') * bnode(q_n, l, v_n, r) * ((v_n < c \wedge tpath(r, q, c, \beta_2, \gamma) * \\
 & \quad subtree(l, \beta_1)) \vee (v_n > c \wedge tpath(l, q, c, \beta_1, \gamma) * subtree(r, \beta_2)))) * subtree(q, \gamma \cup \{c\}) \\
\Rightarrow & (\beta_n \cap \gamma = \emptyset) \wedge ((\beta_n = \{v_n\} \cup \beta_1 \cup \beta_2) \wedge tpath(p, q_n, c, \beta', \gamma') * (bnode(q_n, l, v_n, r) * \\
 & \quad ((upper(\beta_1, v_n) \wedge lower(\beta_2 \cup \gamma \cup \{c\}, v_n) \wedge r = q \wedge \beta_2 = \emptyset \wedge tpath(q, q, c, \beta_2, \gamma) * \\
 & \quad subtree(l, \beta_1)) \vee (upper(\beta_1 \cup \gamma \cup \{c\}, v_n) \wedge lower(\beta_2, v_n) \wedge l = q \wedge \beta_1 = \emptyset \wedge \\
 & \quad tpath(q, q, c, \beta_1, \gamma) * subtree(r, \beta_2))) * subtree(q, \gamma \cup \{c\}))) \\
\Rightarrow & (\beta' \cap \gamma' = \emptyset) \wedge tpath(p, p_n, c, \beta', \gamma') * subtree(p_n, \beta_n \cup \gamma \cup \{c\}) \\
= & (\beta' \cap \gamma' = \emptyset) \wedge tpath(p, p_n, c, \beta', \gamma') * subtree(p_n, \gamma' \cup \{c\}) \\
\Rightarrow & subtree(p, \beta' \cup \beta_n \cup \gamma \cup \{c\}) \\
= & subtree(p, \beta \cup \gamma \cup \{c\})
\end{array}
$$

By the induction on the length of the trace, we have the conclusion. □

Based on already well specified method specifications, specification predicates and their properties in lemmas above, we are ready to verify the correctness of class $TreeSet$ using our inference rules. For each method, we give the proof for its body command. Firstly, we prove constructors $BNode.BNode$ and $TreeSet.TreeSet$ (in **Fig. 26**) as follows.

Proving $BNode.BNode$**:**

$\{\mathsf{raw}(\mathbf{this}, BNode)\}$
$\mathbf{this}.val = v;\ \ \mathbf{this}.left = \mathbf{null};$
$\quad \mathbf{this}.right = \mathbf{null};$
$\{\mathbf{this}.val \mapsto v * \mathbf{this}.left \mapsto \mathsf{rnull} *$
$\quad \mathbf{this}.right \mapsto \mathsf{rnull}\}$
$\{bnode(\mathbf{this}, \mathsf{rnull}, v, \mathsf{rnull})\}$

Proving $TreeSet.TreeSet$**:**

$\{\mathsf{raw}(\mathbf{this}, TreeSet)\}$
$\mathbf{this}.rt = \mathbf{null};$
$\{\mathbf{this}.rt \mapsto \mathsf{rnull} * \mathbf{emp}\}$
$\{\mathbf{this}.rt \mapsto \mathsf{rnull} * subtree(\mathsf{rnull}, \emptyset)\}$
$\{set(\mathbf{this}, \emptyset)\}$

Fig. 26. Proofs for $BNode.BNode$ and $TreeSet.TreeSet$

Then for methods $empty$, add, $contain$ in class $TreeSet$, their specifications are also inherited from interface Set. Therefore, as we do for methods in $ListSet$ similarly, we should do method body verifications with their inherited specifications by rule [S-MTHD2]. As follows, we give processes to verify these method bodies meeting their specifications in detail.

- *TreeSet.empty* **with its specification.**

$\{b = \mathsf{rfalse} \wedge p = \mathsf{rnull} \wedge set(\mathbf{this}, \alpha)\}$
$p = \mathbf{this}.rt;$
$\{\exists r_p \cdot\ b = \mathsf{rfalse} \wedge p = r_p \wedge\ \mathbf{this}.rt \mapsto\ r_p * subtree(r_p, \alpha)\}$
if $(p == \mathbf{null})$
$\{b = \mathsf{rfalse} \wedge p = \mathsf{rnull} \wedge \alpha = \emptyset \wedge \mathbf{this}.rt \mapsto \mathsf{rnull} * \mathbf{emp}\}$
$\{b = \mathsf{rfalse} \wedge p = \mathsf{rnull} \wedge \alpha = \emptyset\}$
$b = \mathbf{true};$
$\{b = \mathsf{rtrue} \wedge \alpha = \emptyset\}$
else
$\{\exists\, r_p, v, r_1, r_2, \beta_1, \beta_2 \cdot (\alpha = \beta_1 \cup \{v\} \cup \beta_2) \wedge upper(\beta_1, v) \wedge lower(\beta_2, v) \wedge$
$b = \mathsf{rfalse} \wedge p = r_p \wedge r_p \neq \mathsf{rnull} \wedge \mathbf{this}.rt \mapsto\ r_p * bnode(r_p, r_1, v, r_2) *$
$subtree(r_1, \beta_1) * subtree(r_2, \beta_2)\}$
$b = \mathbf{false};$
$\{b = \mathsf{rfalse} \wedge p \neq \mathsf{rnull} \wedge \alpha \neq \emptyset\}$
$\{b = \mathsf{rfalse} \wedge \alpha \neq \emptyset\}$
return $b;$
$\{set(\mathbf{this}, \alpha) \wedge ((\alpha = \emptyset \wedge \mathsf{res} = \mathsf{rtrue}) \vee (\alpha \neq \emptyset \wedge \mathsf{res} = \mathsf{rfalse}))\}$
$\{set(\mathbf{this}, \alpha) \wedge \mathsf{res} = (\alpha = \emptyset)\}$

- *TreeSet.add* **with its specification.**

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge t = \mathsf{rnull} \wedge v = 0 \wedge set(\mathbf{this}, \alpha)\}$
$v = 0;$
$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge v = 0 \wedge set(\mathbf{this}, \alpha)\}$
$p = \mathbf{this}.rt;\ q = \mathbf{null};$
$\{\exists r_p, r_q \cdot p = r_p \wedge q = r_q \wedge r_q = \mathsf{rnull} \wedge v = 0 \wedge \mathbf{this}.rt \mapsto r_p * subtree(r_p, \alpha)\}$
while $(p\,!{=}\mathbf{null})\{$
$\{\exists r_p, r_q \cdot p = r_p \wedge q = r_q \wedge ((\mathbf{this}.rt \mapsto r_p \wedge r_q = \mathsf{rnull} \wedge subtree(r_p, \alpha)) \vee$
$(\exists \beta, \gamma, v, \gamma_1, \gamma_2 \cdot r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge$
$upper(\gamma_1, v) \wedge lower(\gamma_2, v) \wedge (tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * bnode(r_q, l, v, r) *$
$subtree(l, \gamma_1) * subtree(r, \gamma_2)) \wedge$
$((c < v \wedge l = r_p) \vee (c > v \wedge r = r_p) \vee (c = v \wedge r_p = \mathsf{rnull})))\}$
$q = p;$
$\{\exists r_p, r_q, v, v', \beta, \beta', \gamma, \gamma', \gamma_1, \gamma_1', \gamma_2, \gamma_2' \cdot p = r_p \wedge q = r_q \wedge r_q = r_p \wedge r_q \neq \mathsf{rnull} \wedge$
$r_p \neq \mathsf{rnull} \wedge (\alpha = \beta' \cup \gamma') \wedge (\beta' \cap \gamma' = \emptyset) \wedge (\gamma' = \gamma_1' \cup \gamma_2' \cup \{v'\}) \wedge$
$(((\beta' = \beta \cup \{v\} \cup \gamma_1) \wedge (\gamma' = \gamma_2) \wedge (v > c)) \vee ((\beta' = \beta \cup \{v\} \cup \gamma_2) \wedge (\gamma' = \gamma_1) \wedge (v < c))) \wedge$
$upper(\gamma_1', v') \wedge lower(\gamma_2', v') \wedge tpath(\mathbf{this}.rt, r_q, c, \beta', \gamma') * bnode(r_q, l', v', r') *$
$subtree(l', \gamma_1') * subtree(r', \gamma_2')\}$
$v = q.val;$
$\{\exists r_p, r_q, v, \beta', \gamma', \gamma_1', \gamma_2' \cdot p = r_p \wedge q = r_q \wedge r_q = r_p \wedge r_q \neq \mathsf{rnull} \wedge r_p \neq \mathsf{rnull} \wedge$
$(\alpha = \beta' \cup \gamma') \wedge (\beta' \cap \gamma' = \emptyset) \wedge (\gamma' = \gamma_1' \cup \gamma_2' \cup \{v\}) \wedge upper(\gamma_1', v) \wedge$
$lower(\gamma_2', v) \wedge tpath(\mathbf{this}.rt, r_q, c, \beta', \gamma') * bnode(r_q, l', v, r') *$
$subtree(l', \gamma_1') * subtree(r', \gamma_2')\}$

$$
\begin{array}{l}
\quad \textbf{if } (v == c) \\
\quad\quad \{\exists r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot p = r_p \wedge q = r_q \wedge r_p = r_q \wedge r_q \neq \mathsf{rnull} \wedge c = v \wedge \\
\quad\quad\quad (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \\
\quad\quad\quad \mathit{lower}(\gamma_2, v) \wedge \mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{bnode}(r_q, l, v, r) * \\
\quad\quad\quad \mathit{subtree}(l, \gamma_1) * \mathit{subtree}(r, \gamma_2)\} \\
\quad\quad p = \mathbf{null}; \\
\quad\quad \{\exists r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot q = r_q \wedge r_q \neq \mathsf{rnull} \wedge p = \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge \\
\quad\quad\quad (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \mathit{lower}(\gamma_2, v) \wedge c = v \wedge \mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \\
\quad\quad\quad \mathit{bnode}(r_q, l, v, r) * \mathit{subtree}(l, \gamma_1) * \mathit{subtree}(r, \gamma_2)\} \\
\quad\quad \{\exists r_q, \beta, \gamma, v \cdot q = r_q \wedge r_q \neq \mathsf{rnull} \wedge p = \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge c = v \wedge \\
\quad\quad\quad v \in \alpha \wedge \mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{subtree}(r_q, \gamma)\} \\
\quad\quad \{\exists v \cdot q \neq \mathsf{rnull} \wedge p = \mathsf{rnull} \wedge c = v \wedge v \in \alpha \wedge \mathit{subtree}(\mathbf{this}.rt, \alpha)\} \\
\quad \textbf{if } (v < c) \\
\quad\quad \{\exists r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot p = r_p \wedge q = r_q \wedge r_q = r_p \wedge r_q \neq \mathsf{rnull} \wedge r_p \neq \mathsf{rnull} \wedge c > v \wedge \\
\quad\quad\quad (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \mathit{lower}(\gamma_2, v) \wedge \\
\quad\quad\quad (\mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{bnode}(r_q, l, v, r) * \mathit{subtree}(l, \gamma_1) * \mathit{subtree}(r, \gamma_2))\} \\
\quad\quad p = q.\mathit{right}; \\
\quad\quad \{\exists r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge c > v \wedge (\alpha = \beta \cup \gamma) \wedge \\
\quad\quad\quad (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \mathit{lower}(\gamma_2, v) \wedge \\
\quad\quad\quad \mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{bnode}(r_q, l, v, r_p) * \mathit{subtree}(l, \gamma_1) * \mathit{subtree}(r_p, \gamma_2)\} \\
\quad \textbf{else} \\
\quad\quad \{\exists r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot p = r_p \wedge q = r_q \wedge r_q = r_p \wedge r_q \neq \mathsf{rnull} \wedge r_p \neq \mathsf{rnull} \wedge c < v \wedge \\
\quad\quad\quad (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \mathit{lower}(\gamma_2, v) \wedge \\
\quad\quad\quad (\mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{bnode}(r_q, l, v, r) * \mathit{subtree}(l, \gamma_1) * \mathit{subtree}(r, \gamma_2))\} \\
\quad\quad p = q.\mathit{left}; \\
\quad\quad \{\exists r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge c < v \wedge (\alpha = \beta \cup \gamma) \wedge \\
\quad\quad\quad (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \mathit{lower}(\gamma_2, v) \wedge \\
\quad\quad\quad \mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{bnode}(r_q, r_p, v, r) * \mathit{subtree}(r, \gamma_2) * \mathit{subtree}(r_p, \gamma_1)\} \\
\} \\
\{\exists r_q, v, \beta, \gamma, \gamma_1, \gamma_2 \cdot q = r_q \wedge ((r_q = \mathsf{rnull} \wedge \alpha = \emptyset) \vee (r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge \\
\quad (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \mathit{lower}(\gamma_2, v) \wedge \\
\quad (\mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{bnode}(r_q, l, v, r) * \mathit{subtree}(l, \gamma_1) * \mathit{subtree}(r, \gamma_2)) \wedge \\
\quad ((v = c) \vee (c \notin \alpha \wedge ((v < c \wedge r = \mathsf{rnull} \wedge \gamma_2 = \emptyset) \vee (v > c \wedge l = \mathsf{rnull} \wedge \gamma_1 = \emptyset))))))\} \\
\textbf{if } (v\,!{=}c)\ \{ \\
\quad \{\exists r_q, v, \beta, \gamma, \gamma_1, \gamma_2 \cdot q = r_q \wedge ((r_q = \mathsf{rnull} \wedge \alpha = \emptyset) \vee (r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge \\
\quad\quad (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \mathit{lower}(\gamma_2, v) \wedge \\
\quad\quad (\mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{bnode}(r_q, l, v, r) * \mathit{subtree}(l, \gamma_1) * \mathit{subtree}(r, \gamma_2)) \wedge \\
\quad\quad c \notin \alpha \wedge ((v < c \wedge r = \mathsf{rnull} \wedge \gamma_2 = \emptyset) \vee (v > c \wedge l = \mathsf{rnull} \wedge \gamma_1 = \emptyset))))\} \\
\quad t = \mathbf{new}\ \mathit{BNode}(c); \\
\quad \{\exists r_t, r_q, v, \beta, \gamma, \gamma_1, \gamma_2 \cdot q = r_q \wedge t = r_t \wedge ((r_q = \mathsf{rnull} \wedge \alpha = \emptyset) \vee (r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge \\
\quad\quad (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge \mathit{upper}(\gamma_1, v) \wedge \mathit{lower}(\gamma_2, v) \wedge c \notin \alpha \wedge \\
\quad\quad (\mathit{tpath}(\mathbf{this}.rt, r_q, c, \beta, \gamma) * \mathit{bnode}(r_q, l, v, r) * \mathit{subtree}(l, \gamma_1) * \mathit{subtree}(r, \gamma_2)) \wedge \\
\quad\quad ((v < c \wedge r = \mathsf{rnull} \wedge \gamma_2 = \emptyset) \vee (v > c \wedge l = \mathsf{rnull} \wedge \gamma_1 = \emptyset)))) \wedge \mathit{bnode}(r_t, \mathsf{rnull}, c, \mathsf{rnull})\} \\
\quad \textbf{if } (q == \mathbf{null}) \\
\quad\quad \{\exists r_t, r_q, \cdot q = r_q \wedge t = r_t \wedge r_q = \mathsf{rnull} \wedge \alpha = \emptyset \wedge \mathit{bnode}(r_t, \mathsf{rnull}, c, \mathsf{rnull})\} \\
\quad\quad \mathbf{this}.rt = t; \\
\quad\quad \{\exists r_q, r_t \cdot t = r_t \wedge q = r_q \wedge r_q = \mathsf{rnull} \wedge \mathbf{this}.rt \mapsto r_t * \mathit{bnode}(r_t, \mathsf{rnull}, c, \mathsf{rnull})\} \\
\quad\quad \{\mathit{set}(\mathbf{this}, \{c\})\}
\end{array}
$$

$\quad$ **else if** $(v < c)$

$\quad\quad \{\exists r_t, r_q, v, \beta, \gamma, \gamma_1, \gamma_2 \cdot q = r_q \wedge t = r_t \wedge r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge c \notin \alpha \wedge$

$\quad\quad\quad (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge (\gamma_2 = \emptyset) \wedge upper(\gamma_1, v) \wedge v < c \wedge tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) *$

$\quad\quad\quad bnode(r_q, l, v, \mathsf{rnull}) * subtree(l, \gamma_1) * bnode(r_t, \mathsf{rnull}, c, \mathsf{rnull})\}$

$\quad\quad q.right = t;$

$\quad\quad \{\exists r_t, r_q, v, \beta, \gamma, \gamma_1 \cdot q = r_q \wedge t = r_t \wedge c \notin \alpha \wedge r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge$

$\quad\quad\quad (\gamma = \gamma_1 \cup \{v\}) \wedge upper(\gamma_1, v) \wedge lower(\{c\}, v) \wedge tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) *$

$\quad\quad\quad bnode(r_q, l, v, r_t) * subtree(l, \gamma_1) * subtree(r_t, \{c\})\}$

$\quad\quad \{\exists r_t, r_q, \beta, \gamma \cdot q = r_q \wedge t = r_t \wedge c \notin \alpha \wedge r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge$

$\quad\quad\quad tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * subtree(r_q, \gamma \cup \{c\})\}$

$\quad\quad \{c \notin \alpha \wedge subtree(\mathbf{this}.rt, \alpha \cup \{c\})\}$

$\quad$ **else if** $(v > c)$

$\quad\quad \{\exists r_t, r_q, v, \beta, \gamma, \gamma_1, \gamma_2 \cdot q = r_q \wedge t = r_t \wedge r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge c \notin \alpha \wedge$

$\quad\quad\quad (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge (\gamma_1 = \emptyset) \wedge lower(\gamma_2, v) \wedge v > c \wedge tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) *$

$\quad\quad\quad bnode(r_q, \mathsf{rnull}, v, r) * subtree(r, \gamma_2) * bnode(r_t, \mathsf{rnull}, c, \mathsf{rnull})\}$

$\quad\quad q.left = t;$

$\quad\quad \{\exists r_t, r_q, v, \beta, \gamma, \gamma_2 \cdot q = r_q \wedge t = r_t \wedge c \notin \alpha \wedge r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge$

$\quad\quad\quad (\gamma = \gamma_2 \cup \{v\}) \wedge lower(\gamma_2, v) \wedge upper(\{c\}, v) \wedge tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) *$

$\quad\quad\quad bnode(r_q, r_t, v, r) * subtree(r, \gamma_2) * subtree(r_t, \{c\})\}$

$\quad\quad \{\exists r_t, r_q, \beta, \gamma \cdot q = r_q \wedge t = r_t \wedge c \notin \alpha \wedge r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge$

$\quad\quad\quad tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * subtree(r_q, \gamma \cup \{c\})\}$

$\quad\quad \{c \notin \alpha \wedge subtree(\mathbf{this}.rt, \alpha \cup \{c\})\}$

$\}$

$\{set(\mathbf{this}, \alpha \cup \{c\})\}$

- *TreeSet.contain* **with its specification.**

$\{p = \mathsf{rnull} \wedge q = \mathsf{rnull} \wedge b = \mathsf{rfalse} \wedge v = 0 \wedge set(\mathbf{this}, \alpha)\}$

$p = \mathbf{this}.rt;\ q = \mathbf{null};\ b = \mathbf{false};\ v = 0;$

$\{\exists r_p, r_q, r_b \cdot p = r_p \wedge q = r_q \wedge b = r_b \wedge r_q = \mathsf{rnull} \wedge$

$\quad r_b = \mathsf{rfalse} \wedge v = 0 \wedge \mathbf{this}.rt \mapsto r_p * subtree(r_p, \alpha)\}$

while $(p\,!{=}\mathbf{null} \wedge b\,!{=}\mathbf{true})\{$

$\quad \{\exists r_b, r_p, r_q \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge ((\mathbf{this}.rt \mapsto r_p \wedge r_q = \mathsf{rnull} \wedge r_b = \mathsf{rfalse} \wedge$

$\quad\quad subtree(r_p, \alpha)) \vee (\exists \beta, \gamma, v, \gamma_1, \gamma_2 \cdot r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge$

$\quad\quad (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge upper(\gamma_1, v) \wedge lower(\gamma_2, v) \wedge r_q \neq \mathsf{rnull} \wedge$

$\quad\quad (tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * bnode(r_q, l, v, r) * subtree(l, \gamma_1) *$

$\quad\quad subtree(r, \gamma_2)) \wedge (r_b = \mathsf{rfalse} \wedge ((c < v \wedge l = r_p) \vee (c > v \wedge r = r_p))) \vee$

$\quad\quad (c = v \wedge r_p = \mathsf{rnull} \wedge r_b = \mathsf{rtrue}))\}$

$\quad q = p;$

$\quad \{\exists r_b, r_p, r_q, v, v', \beta, \beta', \gamma, \gamma', \gamma_1, \gamma_1', \gamma_2, \gamma_2' \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge r_q = r_p \wedge$

$\quad\quad r_q \neq \mathsf{rnull} \wedge r_p \neq \mathsf{rnull} \wedge r_b = \mathsf{rfalse} \wedge (\alpha = \beta' \cup \gamma') \wedge (\beta' \cap \gamma' = \emptyset) \wedge$

$\quad\quad (((\beta' = \beta \cup \{v\} \cup \gamma_1) \wedge (\gamma' = \gamma_2) \wedge (v > c)) \vee ((\beta' = \beta \cup \{v\} \cup \gamma_2) \wedge (\gamma' = \gamma_1) \wedge (v < c))) \wedge$

$\quad\quad (\gamma' = \gamma_1' \cup \gamma_2' \cup \{v'\}) \wedge upper(\gamma_1', v') \wedge lower(\gamma_2', v') \wedge tpath(\mathbf{this}.rt, r_q, c, \beta', \gamma') *$

$\quad\quad bnode(r_q, l', v', r') * subtree(l', \gamma_1') * subtree(r', \gamma_2')\}$

$\quad v = q.val;$

$\quad \{\exists r_b, r_p, r_q, v, \beta', \gamma', \gamma_1', \gamma_2' \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge r_q = r_p \wedge r_q \neq \mathsf{rnull} \wedge$

$\quad\quad r_p \neq \mathsf{rnull} \wedge r_b = \mathsf{rfalse} \wedge (\alpha = \beta' \cup \gamma') \wedge (\beta' \cap \gamma' = \emptyset) \wedge (\gamma' = \gamma_1' \cup \gamma_2' \cup \{v\}) \wedge$

$\quad\quad upper(\gamma_1', v) \wedge lower(\gamma_2', v) \wedge tpath(\mathbf{this}.rt, r_q, c, \beta', \gamma') * bnode(r_q, l', v, r') *$

$\quad\quad subtree(l', \gamma_1') * subtree(r', \gamma_2')\}$

$\mathbf{if}\ (v == c)$
$\{\exists r_b, r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge r_p = r_q \wedge r_q \neq \mathsf{rnull} \wedge$
$r_b = \mathsf{rfalse} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge upper(\gamma_1, v) \wedge$
$lower(\gamma_2, v) \wedge (tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * bnode(r_q, l, v, r) * subtree(l, \gamma_1) *$
$subtree(r, \gamma_2)) \wedge c = v\}$
$b = \mathbf{true};$
$\{\exists r_b, r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge c = v \wedge r_q \neq \mathsf{rnull} \wedge r_p = r_q \wedge$
$r_b = \mathsf{rtrue} \wedge subtree(\mathbf{this}.rt, \alpha))\}$
$\mathbf{if}\ (v < c)$
$\{\exists r_b, r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge r_p = r_q \wedge r_q \neq \mathsf{rnull} \wedge$
$r_b = \mathsf{rfalse} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge upper(\gamma_1, v) \wedge$
$lower(\gamma_2, v) \wedge (tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * bnode(r_q, l, v, r) * subtree(l, \gamma_1) *$
$subtree(r, \gamma_2)) \wedge c > v\}$
$p = q.right;$
$\{\exists r_b, r_p, r_q, v, \beta, \gamma, \gamma_1, \gamma_2 \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge r_b = \mathsf{rfalse} \wedge$
$c > v \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge upper(\gamma_1, v) \wedge$
$lower(\gamma_2, v) \wedge tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * bnode(r_q, l, v, r_p) * subtree(l, \gamma_1) *$
$subtree(r_p, \gamma_2)\}$
$\mathbf{else}$
$\{\exists r_b, r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge r_p = r_q \wedge r_q \neq \mathsf{rnull} \wedge$
$r_b = \mathsf{rfalse} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge upper(\gamma_1, v) \wedge$
$lower(\gamma_2, v) \wedge (tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * bnode(r_q, l, v, r) * subtree(l, \gamma_1) *$
$subtree(r, \gamma_2)) \wedge c < v\}$
$p = q.left;$
$\{\exists r_b, r_p, r_q, \beta, \gamma, v, \gamma_1, \gamma_2 \cdot b = r_b \wedge p = r_p \wedge q = r_q \wedge r_q \neq \mathsf{rnull} \wedge r_b = \mathsf{rfalse} \wedge$
$c < v \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge upper(\gamma_1, v) \wedge$
$lower(\gamma_2, v) \wedge tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * bnode(r_q, r_p, v, r) * subtree(r_p, \gamma_1) *$
$subtree(r, \gamma_2)\}$
$\}$
$\{\exists r_q, r_b, v, \gamma, \gamma_1, \gamma_2 \cdot q = r_q \wedge b = r_b \wedge ((r_q = \mathsf{rnull} \wedge \alpha = \emptyset \wedge r_b = \mathsf{rfalse}) \vee$
$(r_q \neq \mathsf{rnull} \wedge (\alpha = \beta \cup \gamma) \wedge (\beta \cap \gamma = \emptyset) \wedge (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \wedge upper(\gamma_1, v) \wedge$
$lower(\gamma_2, v) \wedge (tpath(\mathbf{this}.rt, r_q, c, \beta, \gamma) * bnode(r_q, l, v, r) * subtree(l, \gamma_1) *$
$subtree(r, \gamma_2)) \wedge ((v = c \wedge r_b = \mathsf{rtrue}) \vee (c \notin \alpha \wedge r_b = \mathsf{rfalse} \wedge$
$((v < c \wedge r = \mathsf{rnull} \wedge \gamma_2 = \emptyset) \vee (v > c \wedge l = \mathsf{rnull} \wedge \gamma_1 = \emptyset))))))\}$

$\{\exists r_b \cdot b = r_b \wedge ((\alpha = \emptyset \wedge r_b = \mathsf{rfalse}) \vee (\alpha \neq \emptyset \wedge ((c \notin \alpha \wedge r_b = \mathsf{rfalse}) \vee$
$(c \in \alpha \wedge r_b = \mathsf{rtrue})) \wedge subtree(\mathbf{this}.rt, \alpha)))\}$
$\mathbf{return}\ b;$
$\{((c \notin \alpha \wedge \mathsf{res} = \mathsf{rfalse}) \vee (c \in \alpha \wedge \mathsf{res} = \mathsf{rtrue})) \wedge set(\mathbf{this}, \alpha)\}$
$\{set(\mathbf{this}, \alpha) \wedge \mathsf{res} = (c \in \alpha)\}$

Until now, we have finished the verification of the correctness of class $TreeSet$ and can conclude that it is also a correct implementation of interface Set.

To reveal how modularity and abstraction our verification framework can do, we illustrate a client code $test$ in **Fig. 27** where interface Set and its declared methods are used, and also prove it only by using method specifications of called methods without reverification of the method bodies.

In the client code $test$, we firstly declare two local variables x, y of type Set, and then instantiate them as two objects while x is of $TreeSet$ and y is of $ListSet$ separately. Later, we make some calls for methods add, $contain$, $empty$ and return the last value.

$\mathbf{Bool}\ test()\ /*\ Client\ */$
$\langle \mathbf{true} \rangle \langle \mathsf{res} = \mathsf{rfalse} \rangle$
$\{\ Set\ x,\ y;\ \mathbf{Bool}\ b_1, b_2;$
$\quad x = \mathbf{new}\ TreeSet();\ y = \mathbf{new}\ ListSet();\ x.add(3);\ y.add(2);$
$\quad b_1 = x.contain(2);\ b_2 = y.empty();\ \mathbf{return}\ b_2;$
$\}$

Fig. 27. Client Code for Set

Using method specifications recorded in interface Set, we intend to check whether calls for methods on objects which are instantiated by implementation classes $ListSet$ and $TreeSet$ separately, will affect each other, and whether should we need to reverify method bodies when call them in the client code. In the following, we will concentrate on these points when do proving step by step.

• **Verification of the Client code**

$\{x = \mathsf{rnull} \wedge y = \mathsf{rnull} \wedge b_1 = \mathsf{rfalse} \wedge b_2 = \mathsf{rfalse}\}$
$x = \mathbf{new}\ TreeSet();\ y = \mathbf{new}\ ListSet();$
$\{\exists\, r_x, r_y \cdot x = r_x \wedge y = r_y \wedge b_1 = \mathsf{rfalse} \wedge b_2 = \mathsf{rfalse} \wedge set(r_x, \emptyset) * set(r_y, \emptyset)\}$
$x.add(3);\ y.add(2);$
$\{\exists\, r_x, r_y \cdot x = r_x \wedge y = r_y \wedge b_1 = \mathsf{rfalse} \wedge b_2 = \mathsf{rfalse} \wedge set(r_x, \{3\}) * set(r_y, \{2\})\}$
$b_1 = x.contain(2);$
$\{\exists\, r_x, r_y \cdot x = r_x \wedge y = r_y \wedge b_1 = \mathsf{rfalse} \wedge b_2 = \mathsf{rfalse} \wedge set(r_x, \{3\}) * set(r_y, \{2\})\}$
$b_2 = y.empty();$
$\{\exists\, r_x, r_y \cdot x = r_x \wedge y = r_y \wedge b_1 = \mathsf{rfalse} \wedge b_2 = \mathsf{rfalse} \wedge set(r_x, \{3\}) * set(r_y, \{2\})\}$
$\mathbf{return}\ b_2;$
$\{\mathsf{res} = \mathsf{rfalse}\}$

During the above verification of client code, we can see that just abstract method specifications recorded rather than specifications specified in implementation classes are needed. Moreover, calling for method add on object $ListSet.y$ does not affect another object $TreeSet.x$, conversely does not yet. After adding an integer value 2 for y and 3 for x, we call method $contain$ on x to judge whether it contains a value 2 and get a result **false** which reveals that object x does not contain 2 exactly. All of these method calls and their results tell us that objects instantiated of $ListSet$ and $TreeSet$ behave independently, thus it truly says, each of these two classes is designed as an independent module. For another interesting thing, we find modularity helps to reduce the load of reverification called method's bodies in the process of client code's proof. We did verify each method with its specification to be well-defined and specified before it is called by client. Then, we constructed each class to be a correct program by recording all its methods. When each class has no coupling with other classes, it is organized as a reusable module by other programs easily. Eventually, we design a modular specification and verification framework with many well specified properties.