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 implemen-
tation 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 de-
veloped 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 ver-
ifying implementation, and the *dynamic* one supports verifying dynamically bound in-
vocations. However, dual specifications are not necessary, and our Verif 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 Verif 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 predicates* 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
inter $I_1 \{ T_1 \ m(\ldots); \ T_2 \ f(\ldots); \}$
inter $I_2 \{ T_3 \ m(\ldots); \ T_3 \ g(\ldots); \}$

class $B : \text{Object} \{ T_1 \ m(\ldots)\{\ldots\} \ T_3 \ g(\ldots)\{\ldots\} \ T_4 \ b(\ldots)\{\ldots\} \ T_5 \ k(\ldots)\{\ldots\} \}$

class $D : B \vdash I_1, I_2 \{ T_1 \ m(\ldots)\{\ldots\} \ T_2 \ f(\ldots)\{\ldots\} \ T_3 \ k(\ldots)\{\ldots\} \ T_5 \ n(\ldots)\{\ldots\} \}$

class $E : \text{Object} \vdash I_1 \{ T_1 \ m(\ldots)\{\ldots\} \ T_2 \ f(\ldots)\{\ldots\} \ T_7 \ p(\ldots)\{\ldots\} \}$

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 $\text{Object}$ 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:

\[
I_1 \ x = \text{new} \ D(.); \ I_2 \ y = (D)x; \ ... \ x.m(.); \ y.m(.);
\]  

(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.,

\[
I_1 \ z = \text{new} \ E(.);
\]

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:

\[
T_1 \ m(\ldots) \langle \varphi \rangle \langle \psi \rangle; \quad T_1 \ m(\ldots) \langle \varphi \rangle \langle \psi \rangle \{ \ldots \}
\]

where \( \varphi \) and \( \psi \) 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 \( \pi_1 \), \( \pi_2 \), 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 \( \pi_1 \), \( \pi_2 \), 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 \( \pi_1 \), \( \pi_2 \), 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:

\[
\begin{align*}
\rho & ::= \text{true} \mid \text{false} \mid r_1 = r_2 \mid r : T \mid r <: T \mid v = r \\
\eta & ::= \text{emp} \mid r_1.a \mapsto r_2 \mid \text{obj}(r, T) \\
\psi & ::= \rho \mid \eta \mid \neg \psi \mid \psi \lor \psi \mid \psi \land \psi \mid \psi \Rightarrow \psi \mid \psi \star \psi \mid \psi \dashrightarrow \psi \mid \exists r \cdot \psi \mid \forall r \cdot \psi
\end{align*}
\]

where \( T \) is a type, \( v \) a variable or constant, \( r_1, r_2 \) references, \( \psi \) an assertion:

- \( \rho \) 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 \( r \).
- \( \eta \) denotes assertions involving heaps: \( \text{emp} \) 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 \( r_2 \); \( \text{obj}(r, T) \) 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 \( r.a \mapsto - \) to assert the existence of objects in heaps.
- \( * \) and \( \star \) are from Separation Logic: \( \psi_1 * \psi_2 \) means the heap can be split into two parts, where \( \psi_1 \) and \( \psi_2 \) hold on each part respectively; \( \psi_1 \mapsto \psi_2 \) means that if a heap satisfying \( \psi_1 \) is added to the heap, the whole heap satisfies \( \psi_2 \).

We allow user-defined predicates to extend vocabulary of the assertion language. A predicate definition takes the form \( p(\tau) : \psi \), where \( p \) is a symbol (predicate name), \( \tau \) are its formal parameters, and \( \psi \) is the body which is an assertion correlated with \( \tau \). 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 \( p(\tau) \) can be used as a basic assertion. We use \( \Psi \) to denote the set of OOSL assertions.

We use \( \psi[r/x] \) (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 \( \exists r'. (v = r' \land r.a \mapsto r) \). Some common abbreviations are:

\[
\begin{align*}
\text{true} & = \exists r' \cdot r.a \mapsto r' \\
\text{false} & = r.a \mapsto r
\end{align*}
\]

We use \( \text{type}(r) \) 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. \( \text{pub} \) is used to announce that a data field or predicate is publicly accessible. Mutation, field accessing, casting, method invocation, and object creation are all taken as special assignments.
We can have a specification \( L \) for a constructor or method in a class or interface. In postcondition \( \psi \) we can use \( \text{old}(e) \) 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 "\( \text{true}\{\text{true}\}". We have user-defined predicates (specification predicates) in our words in the language. \( P \) denotes a predicate definition including a body \( \psi \). We ask for that \text{this} is written explicitly as the first parameter to denote the current object. The non-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 \( I \), and inherit a class \( B \). A sub-interface should not redefine the same methods of its super-interfaces. As in Java, each class has a superclass, possibly \text{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 \( I_G = (\Delta_G, \Theta_G, \Pi_G, \Phi_G) \), where \( \Delta \) is the typing environment recording structural information of declarations; \( \Theta \) is a method lookup environment mapping \( C, m \) to its body; \( \Pi \) records method specifications; and \( \Phi \) 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 \( (T, T') \in \Delta \text{super} \) to mean that \( T' \) is a direct supertype of \( T \). We often omit \( \Delta \). In the example of Fig. 1, we have \( \text{super}(D, I_1), \text{super}(D, I_2), \) and \( \text{super}(D, B) \). We use \( \text{super}(C) \) 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 \( \text{super} \)). 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-
I specifications. We write \( [\text{superclass} \text{ have specifications for } m] \) have multiple specifications \( \pi \) in class \( C \) construction rules of specification predicate and specification environment in Fig. 3.

If it is not explicitly specified, \( \Theta \) to take their bodies. If \( m \) from the superclass. For a type \( T \) 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 (pub or not) with the body, and use \( \Phi(T, p) \) to take \( p \)’s definition in \( T \) with the 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 \( \langle \phi \rangle \langle \psi \rangle \in \Pi(T, m) \) when \( \langle \phi \rangle \langle \psi \rangle \) is a specification of \( m \) in \( T \), and \( \Pi(T, m) = \langle \phi \rangle \langle \psi \rangle \) when \( \langle \phi \rangle \langle \psi \rangle \) 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 \( \Phi \), [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 \( \text{undef} \) for \( p \)'s body is recorded for \( I \). Notice that, \( \text{undef} \) 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 \( p \)'s definition in \( B \) is also recorded for \( C \).

For specification environment \( \Pi \), 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 \( \psi \) holds in method \( m \) of class \( C \) under \( \Gamma \). Clearly, here \( \psi \) can only be a state-independent assertion. We use \( \Gamma, C, m \vdash \{ \varphi \} e \{ \psi \} \) to denote the fact that command \( e \) in method \( m \) satisfies the specification consisting of precondition \( \varphi \) and postcondition \( \psi \), and \( \Gamma \vdash \{ \varphi \} C.C \{ \psi \} \) denotes that constructor \( C \) is correct with respect to specification \( \langle \varphi \rangle \langle \psi \rangle \) under \( \Gamma \). 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 \( \Gamma, C \), iff there exists an assertion \( R \) which is free of the program variables, such that \( \Gamma, C \vdash (\varphi_1 \Rightarrow \varphi_2 * R) \land (\psi_2 * R \Rightarrow \psi_1) \). 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. \( \square \)

Liskov [13] defined the condition for specification refinement as \( \varphi_1 \Rightarrow \varphi_2 \land \psi_2 \Rightarrow \psi_1 \). Above definition is its extension by taking the storage extension into account. This definition follows also the nature refinement order proposed in [8].
Interfaces, their inheritance, and the behavioral subtyping relation also correlate with verifications, which we call static verification. For an interface \( I \) with superinterface \( I' \), 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 \( FV(R) \) is the set of all program variables (including internal res) in assertion \( R \), and \( MD(e) \) is the set of variables modified by command \( c \). [H-THIS] is simply a type assertion. [H-OLD] says that if assertion \( \psi' \) is provable in the pre-state, then \( \psi'\langle\text{old}(e)/c\rangle \) 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.

---

**Fig. 4. Basic Inference Rules**
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 \( D \), \( p(r, r') \) can be unfolded to the body of \( p \). [H-SPRE] is for the static binding, in that case, \( D.p(r, r') \) is unfolded to its definition in \( D \), where \( \text{fix}(D, \psi) \) gives the instantiation of \( \psi \) in \( D \):

\[
\text{fix}(D, \psi) = \begin{cases} 
-\text{fix}(D, \psi'), & \text{if } \psi = -\psi' \\
\text{fix}(D, \psi_1) \otimes \text{fix}(D, \psi_2), & \text{if } \psi = \psi_1 \otimes \psi_2 \\
\exists r \cdot \text{fix}(D, \psi'), & \text{if } \psi = \exists r \cdot \psi' \\
D.q(\text{this}, \tau), & \text{if } \psi = q(\text{this}, \tau) \\
\psi, & \text{otherwise.}
\end{cases}
\]

Here \( \otimes \) can be \( \lor, \ast, \) or \( -\ast \). Intuitively, \( \text{fix} \) 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 \( \psi \) according to a given class. Notice here in unfolding \( D.q(r, r') \), we use \( \text{fix}(D, \psi) \) 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 \( \tau \) are not free in \( \varphi, \psi \). 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 \( \Pi \) to type set, thus \( \Pi(\text{super}(C))(m) \) 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 \( \text{super}(C) \) 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.
C has specification for m, \( \Theta(C.m) = \lambda(\tau)\{\text{var } \tau; c\} \), \( \Pi(C.m) = \langle \varphi \rangle(\psi) \)

\[
\begin{align*}
\Gamma, C, m \vdash \{\text{this}: C \land \tau = \tau' \land \varphi[r/\tau']\} \land \psi[r/\tau]\}
\quad & \land \langle \varphi' \rangle(\psi') \in \Pi(\text{super}(C))(m) \land \Gamma, C \vdash \langle \varphi' \rangle(\psi') \subseteq \langle \varphi \rangle(\psi)
\end{align*}
\]

\[\Gamma \vdash \{\varphi\} C.m(\psi)\]

C defines m without specification, \( \Theta(C.m) = \lambda(\tau)\{\text{var } \tau; c\} \)

\[\forall \langle \varphi \rangle(\psi) \in \Pi(C.m) \land \Gamma, C, m \vdash \{\text{this}: C \land \tau = \tau' \land \varphi[r/\tau']\} \land \psi[r/\tau]\}
\quad \Gamma, C, m \vdash \Pi(C.m) \]

C inherits D.m, \( \forall \langle \varphi \rangle(\psi) \in \Pi(C.m) \land \Gamma, C \vdash \langle \varphi \rangle(\psi) \subseteq \langle \text{fix}(D, \varphi) \rangle(\text{fix}(D, \psi))\)

\[\forall I \in \text{super}(C) \land \Pi(I.m) = \langle \varphi' \rangle(\psi') \land \Gamma, C \vdash \langle \varphi' \rangle(\psi') \subseteq \Pi(C.m)\]

\[\Gamma, C, m \vdash \Pi(C,m)\]

\[\Pi(C,C) = \langle \varphi \rangle(\psi), \quad \Theta(C.C) = \lambda(\tau)\{\text{var } \tau; c\} \]

\[\Gamma, C, C \vdash \{\tau = \tau \land \varphi[r/\tau']\text{raw(this, } C) \land \psi[r/\tau]\} \land C.C(\psi) \quad \Gamma \vdash \{\varphi\} C.C(\psi)\]

\[\Gamma, C, m \vdash v : T, \quad \langle \varphi \rangle(\psi) \in \Pi(T.n)\]

\[\Gamma, C, m \vdash \{v = v \land e = v' \land \varphi[r/\tau']\text{this, } \tau, x\} \land \psi[r/\tau, x/\text{this, } \tau, \text{res}]\]

\[\Pi(C'.C') = \langle \varphi \rangle(\psi)\]

\[\Gamma, C', m \vdash \{e = v' \land \varphi[r/\tau']\} \land \psi[r/\tau, x/\text{this, } \tau, \text{res}]\]

**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 \text{fix}(D, \bullet) tofix 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 \( \pi \subseteq \{\pi_i\} \) is defined as \( \exists i : \pi \subseteq \pi_i \).

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

\[
\text{raw}(r, N) \equiv \begin{cases} 
\text{obj}(r, N), & N \text{ has no field } \\
\{v : N \land (r.a_1 \mapsto \text{nil}) \land \cdots \land (r.a_k \mapsto \text{nil})\}, & \text{fields of } N \text{ is } a_1, \ldots, a_k
\end{cases}
\]

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:

$$\Theta(C,m) = \lambda(\overline{x}; c) \\
\exists \langle \varphi \rangle \langle \psi \rangle \in \Pi(C,m) \bullet ((\forall \langle \varphi' \rangle \langle \psi' \rangle \in \Pi(C,m) \bullet \Gamma, C \vdash \langle \varphi' \rangle \langle \psi' \rangle \sqsubseteq \langle \varphi \rangle \langle \psi \rangle) \\
\wedge \Gamma, C, m \vdash \{ \text{this: } C \wedge x = v \wedge y = nil \wedge \varphi[\overline{v}] \} \ c\{\psi[\overline{v}]\} \}$$

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(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(this, v) records both current value $v$ in $x$ and some unconcerned value for $y$, and ReCell.bak(this, v) records
Proofs for Some Simple Cases

For ReCell.set and 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 ReCell.undo is simple and given also in Fig. 7. For ReCell.set, we need

v as the value of y and leaves value of x unconcerned. Because ReCell.set is not explicitly specified, it inherits two specifications from Undoable.set and Cell.set. In the same way, ReCell.get inherits a specification from Cell.get; and ReCell.undo inherits a specification from 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 ReCell.set and 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 ReCell.undo is simple and given also in Fig. 7. For ReCell.set, we need
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:

\[
\Gamma, \text{ReCell} \vdash (\text{cell}(\text{this}, b) \land c = 0) \quad (\text{cell}(\text{this}, b) \land c = 0)
\]
\[
\{c = 0 \land \text{this}.x \leftarrow b \land \text{this}.y \leftarrow c\} \quad \{c = 0 \land \text{this}.x \leftarrow c\}
\]
\[
\{c = b \land \text{this}.x \leftarrow b \land \text{this}.y \leftarrow c\} \quad \{c = b \land \text{this}.x = \text{this}.y\}
\]
\[
\{\text{this}.x = v\} \quad \{\text{this}.x = v\}
\]
\[
\{\text{this}.x \leftarrow v \land \text{this}.y \leftarrow b\} \quad \{\text{this}.x \leftarrow v\}
\]
\[
\{\text{this}.x \leftarrow v \land \text{this}.y \leftarrow b\} \quad \{\text{cell}(\text{this}, v)\}
\]

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

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

\[
\Gamma, \text{ReCell} \vdash (\text{cell}(\text{this}, v) \rightarrow (\text{fix}(\text{Cell}, \text{cell}(\text{this}, v))) \ast R) \land (\text{fix}(\text{Res}, v) \ast R \rightarrow (\text{Res} = v))
\]
\[
\Rightarrow (\text{cell}(\text{this}, v) \rightarrow \text{Cell}(\text{this}, v) \ast R) \land (\text{Cell}(\text{res} = v) \ast R \rightarrow (\text{res} = v))
\]
\[
\Rightarrow ((\text{this}.x \leftarrow v \land \text{this}.y \leftarrow c) \rightarrow (\text{this}.x \leftarrow v \land R)) \land ((\text{res} = v) \rightarrow (\text{res} = v))
\]
functions between the classes with the interfaces which they implement. We suggest
version of Separation Logic, named OOSL, is used for specifying and reasoning VeriJ
verification. The OO language VeriJ used here takes the pure reference semantics. A
develop here a formal framework which offers modularity for both specification and
proving integrity of our framework. Let us consider a simple example of a client

```
Bool cell_test()
{true} (res = true)
{int c1, c2;
  bool b = false;
  Cell t1;
  Undoable t2;
  t1 = new ReCell();
  t2 = new Undoable();
  t1.set(5);
  c1 = t1.get();
  t2.set(3);
  c2 = t2.get();
if (c1 == c2) b = true;
return b;
}
```

Let “R = this.y → _”, 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 t3 is declared and assigned the same object
by casting t1 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

6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.
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 \( s = (\sigma, O) \in \text{State} \) consists of a store and a heap:

\[
\text{Store} \equiv \text{Name} \rightarrow_{\text{fin}} \text{Ref}, \quad \text{Heap} \equiv \text{Ref} \rightarrow_{\text{fin}} \text{Name} \rightarrow_{\text{fin}} \text{Ref} \\
\text{State} \equiv \text{Store} \times \text{Heap}
\]

\text{Name} is an infinite set of names, with specially \text{true}, \text{false}, \text{null} for boolean constants and null value respectively. \text{Type} is an infinite set of types, where \text{Object}, \text{Null}, \text{Bool}, \text{Int} \in \text{Type}.

\text{Ref} is an infinite set of references, i.e., object identities. It contains three constants: \text{rtrue}, \text{rfalse} refer to the two \text{Bool} objects, and \text{rnul} for. For any \( \sigma \in \text{Store} \), we assume \( \sigma_{\text{true}} = \text{rtrue}, \sigma_{\text{false}} = \text{rfalse} \) and \( \sigma_{\text{null}} = \text{rnul} \). 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 \text{Name}, \text{Type} and \text{Ref}.

Because references in \text{Ref} are atomic, we assume two primitive functions:\(^1\)

\begin{itemize}
  \item \text{eqref} : \text{Ref} \rightarrow \text{Ref} \rightarrow \text{bool}, justifies whether two references are the same, i.e. for any \( r_1, r_2 \in \text{Ref} \), \text{eqref}(r_1, r_2) \text{iff } r_1 \text{ is same to } r_2.
  
  \item \text{type} : \text{Ref} \rightarrow \text{Type} decides the type of the object referred by some reference. We define \( \text{type}(\text{rtrue}) = \text{dtype}(\text{rfalse}) = \text{Bool} \), and \( \text{dtype}(\text{rnul}) = \text{Null} \).
\end{itemize}

A program state \( s = (\sigma, O) \in \text{State} \) 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 \text{Ref} associated with objects, or a subset of \text{Ref} \times \text{Name} associated with values. We use \( \text{dom} O \) to denote the domain of \( O \), and define \( \text{dom}_2 O \equiv \{(r, a) \mid r \in \text{dom} O, a \in \text{dom} O(r)\} \) for the second case.

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

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

\begin{itemize}
  \item \( \forall (r, a) \in \text{dom}_2 O \cdot a \in \text{fields}(\text{type}(r)) \wedge \text{type}(O(r)(a)) \prec \cdot \text{fdtypes}(\text{type}(r))(a), \) and
  
  \item \( \forall r \in \text{dom} O \cdot \text{fields}(\text{type}(r)) = \emptyset \lor (\text{fields}(\text{type}(r)) \cap \text{dom} O(r) \neq \emptyset) \). \hfill \Box
\end{itemize}

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 \( t \in \text{Type} \) and \( id \in \mathbb{N} \), and define \text{eqref} as the pair equality, and \text{type}(r) = r.\text{first}.

\(^2\) Please pay attention that \text{Ref} \rightarrow \text{Name} \rightarrow \text{Ref} \) is very different from \text{Ref} \times \text{Name} \rightarrow \text{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.
Here is a sequence of definitions.

The storage model defined above, with the definition for the separation concept, gives us both the 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 $A$:

$$A ::= \varepsilon \mid p(\tau) \triangleq \varphi, A$$

Here $\varepsilon$ denotes the empty environment, and $A$ is a sequence of definitions.

As the well-formedness, the body assertion $\psi$ of any definition in $A$ cannot use symbols which is not defined in $A$.

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

$$(O_1 \oplus O_2)(r) \triangleq \begin{cases} O_1(r) \cup O_2(r) & \text{if } r \in \text{dom } O_2 \\ O_1(r) & \text{otherwise} \end{cases}$$

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:

$$O_1 \perp O_2 \triangleq \forall r \in \text{dom } O_1 \cap \text{dom } O_2 \cdot (O_1(r) \neq \emptyset \land O_2(r) \neq \emptyset \land \text{dom } (O_1(r)) \cap \text{dom } (O_2(r)) = \emptyset).$$

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 $O_1 \ast O_2$ 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 $A$:
ψ in Λ is syntactically monotone. 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 J. We define the semantics of assertions as a function \( M_\Lambda : \Psi \rightarrow P(\text{State}) \) based on this model by the rules listed in Fig. 10.

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

\[
(\sigma, O) \models \varphi \quad \text{iff} \quad (\sigma, O) \in M(\varphi).
\]

## 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 queues. 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 next reference. We pack its structure using a specification predicate \( \text{node}(\text{this}, v, n) \), which asserts that a Node object holds v and n as its value and the next reference. Note that this predicate is \( \text{pub} \), that means its definition is available everywhere. Node has only a constructor with its specification.

---

3 For definition \( p(r) \models \varphi \), every symbol occurred in \( \psi \) 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, null))
  { this.val = c; this.nxt = null; }
}
inter DQ1{
  def queue(this, α);
  Int get_tail() (queue(this, β :: [α]))
  (res = a ∧ queue(this, β :: [α]));
}
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 :: [α]))
  {res = a ∧ queue(this, β :: [α]);
  }
}

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 […] 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(this, α) states that field hd of a Queue object refers to a single linked list recording a value sequence [0] :: α (Here we assume that 0 is the value of the head node.) Here we have another predicate listseg(this, r1, r2, α), which is given in the common Separation Logic style. It specifies a single linked list segment between r1 and r2, 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, []));
  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))
  { 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, null));
  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,
we verify that its body satisfies the specification, as shown in Fig. 14. 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 \( (P) \vdash (Q) \subseteq (P) \vdash (Q) \). This is trivially true because the two specifications are the same.
\{p = rnull \land q = rnull \land t = rnull \land queue(this, \alpha)\}
\{p = this.hd; q = p.nxt; \}
\{\exists p, \exists r_q \cdot p = r_p \land q = r_q \land t = rnull \land this.hd \rightarrow r_p \ast node(r_p, 0, r_q) \ast listseg(this, r_q, rnull, \alpha)\}
while (q !- null) \{
\{\exists p, \exists r_q, \exists a, \exists \beta, \gamma \cdot p = r_p \land q = r_q \land r_q \neq rnull \land ([0] :: \alpha = \beta :: [a] :: \gamma) \land listseg(this, this.hd, r_p, \beta) \ast node(r_p, a, r_q) \ast listseg(this, r_q, rnull, \gamma)\}
p = q; q = p.nxt; \}
\{\exists p, \beta, a \cdot p = r_p \land q = rnull \land t = rnull \land ([0] :: \alpha = \beta :: [a]) \land listseg(this, this.hd, r_p, \beta) \ast node(r_p, a, rnull)\}
t = new Node(c); p.next = n;
\{\exists p, \exists r_q, \exists a, \exists p = r_p \land t = r1 \land ([0] :: \alpha = \beta :: [a]) \land listseg(this, this.hd, r_p, \beta) \ast node(r_p, a, r1) \ast node(r1, c, rnull)\}
\{listseg(this, this.hd, rnull, [0] :: \alpha :: [c])\}
\{queue(this, \alpha :: [c])\}

\{p = rnull \land q = rnull \land c = 0 \land queue(this, [a] :: \beta)\}
\{p = this.hd; q = p.next; \}
\{\exists p, \exists r_q \cdot p = r_p \land q = r_q \land c = 0 \land this.hd \rightarrow r_p \ast node(r_p, 0, r_q) \ast listseg(this, r_q, rnull, [a] :: \beta)\}
if q !- null \{
\{\exists p, \exists r_q, \exists a, \exists p = r_p \land q = r_q \land r_q \neq rnull \land c = 0 \land this.hd \rightarrow r_p \ast node(r_p, 0, r_q) \ast listseg(this, r_q, rnull, [a] :: \beta)\}
\{c = q.val; \}
\{\exists p, \exists r_q, \exists r \cdot p = r_p \land q = r_q \land r_q \neq rnull \land c = \land this.hd \rightarrow r_p \ast node(r_p, 0, r_q) \ast node(r_q, c, r) \ast listseg(this, r, rnull, \beta)\}
q = q.next; p.next = q;
\{\exists p, \exists r_q, \exists r \cdot p = r_p \land q = r_q \land c = a \land this.hd \rightarrow r_p \ast node(r_p, 0, r_q) \ast node(r_q, c, r) \ast listseg(this, r, rnull, \beta)\}
\}
\{\exists r \cdot c = a \land queue(this, \beta) \ast node(r, c, rnull)\}
return c;
\{res = a \land queue(this, \beta) \ast true\}

\{p = rnull \land q = rnull \land c = 0 \land queue(this, [a] :: \beta)\}
\[p = this.hd; q = p.next; \]
\{\exists p, \exists r_q \cdot p = r_p \land q = r_q \land c = 0 \land this.hd \rightarrow r_p \ast node(r_p, 0, r_q) \ast listseg(this, r_q, rnull, [a] :: \beta)\}
if q !- null \{
\{\exists p, \exists r_q, \exists a, \exists p = r_p \land q = r_q \land r_q \neq rnull \land c = 0 \land this.hd \rightarrow r_p \ast node(r_p, 0, r_q) \ast listseg(this, r_q, rnull, [a] :: \beta)\}
\{c = q.val; \}
\{\exists p, \exists r_q, \exists r \cdot p = r_p \land q = r_q \land r_q \neq rnull \land c = a \land this.hd \rightarrow r_p \ast node(r_p, 0, r_q) \ast node(r_q, c, r) \ast listseg(this, r, rnull, \beta)\}
\{\exists r \cdot c = a \land queue(this, [a] :: \beta)\}
return c;
\{res = a \land queue(this, [a] :: \beta)\}

Fig. 16. Verifying Queue.enqueue, Queue.dequeue, Queue.peek
Proving $DQueue.\ DQueue$:
\[
\{\text{raw}(this, DQueue)\}
\]
Node $t = \text{new } Node(0);\n\{\exists r_t, t = r_t \land \text{raw}(this, DQueue) \land \text{node}(r_t, 0, \text{null})\}
\]
\[
\text{this}.\hd = \text{this}.\tl = t;
\{\exists r_t, t = r_t \land \text{this}.\hd \rightarrow r_t \land \text{this}.\tl \rightarrow r_t \land \text{node}(r_t, 0, \text{null})\}
\]
\[
\{\text{listseg}(\text{this}, r_t, r_t, []) \land \text{node}(r_t, 0, \text{null})\}
\]
\[
\{\text{queue}(\text{this}, [])\}
\]

Proving $DQueue.\ enqueue$:
\[
\{p = \text{null} \land t = \text{null} \land \text{queue}(this, \alpha)\}
\]
\[
p = \text{this}.\tl;\ t = \text{new } Node(c);\n\{\exists r_p, r_t, \beta, a \cdot p = r_p \land t = r_t \land (([] : \alpha = \beta : [a]) \land \text{node}(r_t, c, \text{null}) \land (\text{this}.\hd \rightarrow r \land \text{this}.\tl \rightarrow r \land \text{listseg}(\text{this}, r, r_p, \beta) \land \text{node}(r_p, a, \alpha))\}
\]
\[
p.\text{next} = t;\ \text{this}.\tl = t;
\{\exists r_p, r_t, \beta, a \cdot p = r_p \land t = r_t \land (([] : \alpha = \beta : [a]) \land (\text{this}.\hd \rightarrow r \land \text{this}.\tl \rightarrow r \land \text{list}(r, r_p, \beta) \land \text{node}(r_p, a, \alpha))\}
\]
\[
\{\text{queue}(\text{this}, \alpha : [\alpha])\}
\]

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

Proving $DQueue.\ get\_tail$:
\[
\{\text{queue}(\text{this}, \beta :: [\alpha])\}
\]
\[
\{\exists r, r' \cdot c = 0 \land \text{this}.\hd \rightarrow r \land \text{this}.\tl \rightarrow r' \land \text{listseg}(\text{this}, r, r', \beta) \land \text{node}(r', a, \text{null})\}
\]
\[
c = \text{tl}.\val;\n\{\exists r, r' \cdot c = a \land \text{this}.\hd \rightarrow r \land \text{this}.\tl \rightarrow r' \land \text{listseg}(\text{this}, r, r', \beta) \land \text{node}(r', a, \text{null})\}
\]
\[
\text{return } c;\n\{\text{res} = a \land \text{queue}(\text{this}, \beta :: [\alpha])\}
\]

Proving $DQueue.\ empty$:
\[
\{p = \text{null} \land q = \text{null} \land b = \text{false} \land \text{queue}(this, \alpha)\}
\]
\[
p = \text{this}.\hd;\ q = \text{this}.\tl;
\{\exists r_q, r_t, \beta, p = r_p \land q = r_q \land b = \text{false} \land ([0] : \alpha = \beta : [0]) \land \text{this}.\hd \rightarrow r_p \land \text{this}.\tl \rightarrow r_q \land \text{listseg}(\text{this}, r_p, r_q, \beta) \land \text{node}(r_q, b, \text{null})\}
\]
\[
\text{if } (p=q)
\{\exists r_p, r_q, r, \beta, p = r_p \land q = r_q \land r_p = r_q \land b = \text{false} \land ([0] : \alpha = \beta : [0]) \land \text{this}.\hd \rightarrow r_p \land \text{this}.\tl \rightarrow r_q \land \text{listseg}(\text{this}, r_p, r_q, \beta) \land \text{node}(r_q, b, \text{null})\}
\]
\[
\{\exists r_p \cdot p = r_p \land b = \text{false} \land \text{this}.\hd \rightarrow r_p \land \text{this}.\tl \rightarrow r_p \land \text{listseg}(\text{this}, r_p, r_p, []) \land \text{node}(r_p, 0, \text{null})\}
\]
\[
b = \text{true};\n\{b = \text{true} \land \text{queue}(\text{this}, [])\}
\]
\[
\text{else}
\{\exists r_p, r_q, r, \beta, p = r_p \land q = r_q \land r_p \neq r_q \land b = \text{false} \land ([0] : \alpha = \beta : [0]) \land \text{this}.\hd \rightarrow r_p \land \text{this}.\tl \rightarrow r_q \land \text{listseg}(\text{this}, r_p, r_q, \beta) \land \text{node}(r_q, b, \text{null})\}
\]
\[
b = \text{false};\n\{\exists r_1, r_2 \cdot p = r_2 \land r_2 \neq \text{null} \land b = \text{false} \land \text{this}.\hd \rightarrow r_1 \land \text{node}(r_1, 0, r_2) \land \text{listseg}(\text{this}, r_2, \text{null}, \alpha)\}
\]
\[
\{\exists r_2 \cdot p = r_2 \land r_2 \neq \text{null} \land b = \text{false} \land \text{queue}(\text{this}, \alpha)\}
\]
\[
\text{return } b;\n\{\text{queue}(\text{this}, \alpha) \land ((\alpha = [] \land \text{res} = \text{true}) \lor (\alpha \neq [] \land \text{res} = \text{false}))\}
\]
\[
\{\text{res} = (\alpha = []) \land \text{queue}(\text{this}, \alpha)\}
\]

Fig. 18. Verifying $DQueue.\ get\_tail$ and $DQueue.\ empty$
However, we need also to prove refinement relation between specifications of \(\text{Queue.dequeue} \) and \(\text{DQueue.dequeue} \). Here the situation is different, because it asks for the proof of:

\[
\Gamma, \text{DQueue} \vdash \langle P \rangle Q \subseteq \text{fix}(\text{Queue}, P) \text{fix}(\text{Queue}, Q)
\]

By the Definition 1, we need to prove that there exists an assertion \(R\) that

\[
\Gamma, \text{DQueue}, \text{dequeue} \vdash (P \Rightarrow \text{fix}(\text{Queue}, P) \ast R) \land (\text{fix}(\text{Queue}, Q) \ast R \Rightarrow Q)
\]

where

\[
P = \text{queue(this, } [a] :: \beta) \text{ and } Q = (\text{res} = a \land \text{queue(this, } \beta) \ast \text{true})
\]

By definition of fix, we have

\[
\Gamma, \text{DQueue}, \text{dequeue} \vdash (P \Rightarrow \text{fix}(\text{Queue}, P) \ast R)
\]

\[
\Leftrightarrow (\text{queue(this, } [a]; \beta) \Rightarrow \text{Queue.queue(this, } [a]; \beta) \ast R)
\]

and

\[
\Gamma, \text{DQueue}, \text{dequeue} \vdash (\text{fix}(\text{Queue}, Q) \ast R \Rightarrow Q)
\]

\[
\Leftrightarrow (\text{Queue.queue(this, } \beta) \ast R \Rightarrow \text{queue(this, } \beta))
\]

So, the key point is to prove

\[
\Gamma, \text{DQueue}, \text{dequeue} \vdash \text{Queue.queue(r, } \beta) \ast R \Leftrightarrow \text{queue(r, } \beta)
\]

Let “\(R = \exists r_1 \cdot r. \text{tl} \rightarrow r_1\)”, we have

\[
\Gamma, \text{DQueue}, \text{dequeue} \vdash
\]

\[
\Leftrightarrow \exists r_1 \cdot r. \text{hd} \rightarrow r_1 \ast \text{listseg(this, } r_1, \text{rnul}, [0] :: \beta) \ast (\exists r_1 \cdot r. \text{tl} \rightarrow r_1)
\]

\[
\Leftrightarrow \exists r_1, r_1 \cdot r. \text{hd} \rightarrow r_1 \ast r_1 \ast r. \text{tl} \rightarrow r_1 \ast \text{listseg(this, } r_1, \text{rnul}, [0] :: \beta)
\]

\[
\Leftrightarrow \text{queue(r, } \beta)
\]

Thus, we have proven that \(\text{DQueue.dequeue}\) meets its specifications.

For \(\text{DQueue.peek}\), we only need to prove the refinement relation of specifications:

\[
\Gamma, \text{DQueue} \vdash \langle P \rangle Q \subseteq \text{fix}(\text{Queue}, P) \text{fix}(\text{Queue}, Q)
\]

It is similar to the above proof for \(\text{DQueue.dequeue}\). We omit its details.

Until now, we have proven that classes \(\text{Node, Queue, DQueue}\) with interfaces \(\text{DQ1, DQ2}\) 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 \text{qtest} which declares a variable \(q1\) of class \(\text{Queue}\) but actually initializes it by an queue object of class \(\text{DQueue}\). Then a new variable \(q2\) of interface \(\text{DQ2}\) 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.
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...
Queue
enqueue();dequeue();get_tail();

DQueue
enqueue();empty();get_tail();

Node
val,next

inter DQ2{
def queue(this, α);
void enqueue(Int c); ⟨···⟩ ⟨···⟩
Int dequeue(); ⟨···⟩ ⟨···⟩
Int get_tail(); ⟨···⟩ ⟨···⟩
}
class DQueue : Queue >> DQ2{
def queue(this, α); · · · ;
Node tl;
DQueue() ⟨···⟩ ⟨···⟩{ · · · }
Bool empty() { · · · }
Int get_tail() { · · · }
void enqueue(Int c) { · · · }

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. Moreover, 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 specifications 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, α) 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 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.
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 *sequences* 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
We will have assertion ListSet set definition body of other methods in ListSet Node each of their bodies meeting its specification. Similar to for listseg its auxiliary predicate Node from Fig. 23 from it. Here, specifications for these methods use the embodied predicates in ListSet rule [H-MTHD2] directly.

Before verifying each method (except the constructor) body meeting its inherited specification from Set rule in our modular verification framework correctly. Actually, for ListSet need to verify that class ListSet and then return the result. The Other methods empty actual structure and implementations of ListSet contain, 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.
Then we conclude that ListSet.empty meets its specification.
Then we conclude that contain meets its specification.
• ListSet.add with its specification. Similarly for its body:

\[
\begin{align*}
\{ & p = \text{null} \land q = \text{null} \land n = \text{null} \land b = \text{false} \land \text{set}(\text{this}, \alpha) \} \\
& p = \text{this}.hd; \\
& \{ s, r_p : p = r_p \land g(\alpha, s) \land q = \text{null} \land n = \text{null} \land b = \text{false} \land \text{this}.hd \rightarrow r_p^* \\
& \quad \text{listseg}(\text{this}, r_p, \text{null}, [0] :: s) \} \\
& q = p.nxt; \\
& \{ s, r_p, r_q : p = r_p \land g(\alpha, s) \land q = r_q \land n = \text{null} \land b = \text{false} \land \text{this}.hd \rightarrow r_q^* \\
& \quad \text{node}(r_p, 0, r_q) \ast \text{listseg}(\text{this}, r_q, \text{null}, s) \} \\
& b = \text{this}.\text{contains}(c); \\
& \{ s, r_p, r_q : p = r_p \land g(\alpha, s) \land q = r_q \land n = \text{null} \land b = \text{false} \land c \notin \alpha \land \text{this}.hd \rightarrow r_q^* \\
& \quad \text{node}(r_p, 0, r_q) \ast \text{listseg}(\text{this}, r_q, \text{null}, s) \} \\
& \text{if } (b == \text{false}) \\
& \{ s, r_p, r_q : p = r_p \land g(\alpha, s) \land q = r_q \land n = \text{null} \land b = \text{false} \land c \notin \alpha \land \text{this}.hd \rightarrow r_q^* \\
& \quad \text{node}(r_p, 0, r_q) \ast \text{listseg}(\text{this}, r_q, \text{null}, s) \} \\
& \{ n = \text{new Node}(c); \\
& \quad \{ s, r_n, r_p, r_q : p = r_p \land g(\alpha, s) \land q = r_q \land n = \text{null} \land b = \text{false} \land c \notin \alpha \land n = r_n \land \text{node}(r_n, c, \text{null})^* \\
& \quad \text{this}.hd \rightarrow r_p^* \ast \text{node}(r_p, 0, r_q) \ast \text{listseg}(\text{this}, r_q, \text{null}, s) \} \\
& \quad n.nxt = q; \\
& \{ s, r_n, r_p, r_q : p = r_p \land g(\alpha, s) \land q = r_q \land n = r_n \land b = \text{false} \land c \notin \alpha \land \text{node}(r_n, c, \text{null})^* \\
& \quad \text{this}.hd \rightarrow r_p^* \ast \text{node}(r_p, 0, r_q) \ast \text{listseg}(\text{this}, r_q, \text{null}, s) \} \\
& \quad p.nxt = n; \\
& \{ s, r_n, r_p, r_q : p = r_p \land g(\alpha, s) \land q = r_q \land n = r_n \land b = \text{false} \land c \notin \alpha \land \text{this}.hd \rightarrow r_q^* \\
& \quad \text{node}(r_p, 0, r_q) \ast \text{node}(r_n, c, r_q) \ast \text{listseg}(\text{this}, r_q, \text{null}, s) \} \\
& \quad \{ s, r_p, c \notin \alpha \land g(\alpha, s) \land \text{this}.hd \rightarrow r_p^* \ast \text{listseg}(\text{this}, r_p, \text{null}, [0] :: [c :: s]) \} \\
& \{ c \in \alpha \land \text{set}(\text{this}, \alpha) \} \lor (c \notin \alpha \land \text{set}(\text{this}, \{c \cup \alpha\})) \\
& \{ c \in \alpha \land \text{set}(\text{this}, \{c \cup \alpha\}) \} \lor (c \notin \alpha \land \text{set}(\text{this}, \{c \cup \alpha\})) \\
& \{ \text{set}(\text{this}, \{c \cup \alpha\}) \}
\end{align*}
\]

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(α, 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 \sup Set {
    BNode rt;
    def set(this, α) :
        \exists r. this.rt = r * subtree(r, α);
    def subtree(r, α) :
        (α = ∅ \land r = null \land emp) \lor
        (∃ r1, r2, b, \beta1, \beta2 \cdot (α = \beta1 ∪ {b} \cup \beta2) \land
        upper(\beta1, b) \land lower(\beta2, b) \land
        bnode(r, r1, b, r2) = subtree(r1, \beta1) \land
        subtree(r2, \beta2));
    def upper(α, b) : \forall x \in (x ∈ α) \land (x < b);
    def lower(α, b) : \forall x \in (x ∈ α) \land (x > b);
    def tpath(p, q, c, β, γ) : (c ∈ β \cup γ) \land
    ((p = q) = ∅ = \emptyset \land emp) \lor (∃ l, r, β1, β2.
    (β = β1 ∪ {v} \cup β2) \land bnode(p, l, v, r) \land
    upper(β1, γ) \land lower(β2, v) \land
    tpath(l, q, c, β1, γ) * subtree(r, \beta2)) \lor
    (upper(β1, v) \land lower(β2, γ) \land \{c\}, v) \land
    tpath(r, q, c, \beta2, 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 \land 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(α, b) asserts that b is smaller than all elements in α. On the other hand, tpath recombinates 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 β1 and right subtree with β2 while b is an upper bound of β1 but a lower bound of β2.

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 β1 and a lower bound of another set β2, then we can know that sets β1 and β2 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.**

\[
\begin{align*}
\text{upper}(\beta_1, v) \land \text{lower}(\beta_2, v) & \Rightarrow \beta_1 \cap \beta_2 = \emptyset \\
\text{upper}(\beta_1, v) & \Rightarrow v \notin \beta_1 \\
\text{lower}(\beta_2, v) & \Rightarrow v \notin \beta_2
\end{align*}
\]

Then, **Lemma 3** is more complicate than **Lemma 2**, which indicates the importance of predicate \( t\text{path} \) and deals with the difficulty we would meet with during verifying program codes in TreeSet.

Informally, as illustrated in Fig. 25, predicate \( t\text{path}(p, q, c, \beta, \gamma) \) 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 \( t\text{path}(p, q, c, \beta, \gamma) \) while \( \text{treeset}(q, \gamma) \) is the rest part to search at some time. The set \( \beta \) 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 \( \gamma \) 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 \( t\text{path} \) 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 \( t\text{path} \), suppose we fail to search \( c \) at \( q \) in the original binary tree, then we get two parts \( t\text{path}(p, q, c, \beta, \gamma) \) and \( \text{treeset}(q, \gamma) \) and add a new node with value \( c \) as one of \( q \)’s children. Later, we should combine the new subtree \( \text{subtree}(q, \gamma \cup \{c\}) \) with the other part \( t\text{path}(p, q, c, \beta, \gamma) \) to form a binary tree which we specify as \( \text{treeset}(p, \beta \cup \gamma \cup \{c\}) \). However, it is indeed a difficult problem. Finally, we specify the right property between \( \text{treeset} \) and \( t\text{path} \) in **Lemma 3** to help solving this difficulty and finishing our prove efficiently.

According to the definition of predicate \( t\text{path} \) 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 \( t\text{path}(p, q, c, \beta, \gamma) \) and \( \text{treeset}(q, \gamma) \) as illustrated in Fig. 25. Value sets \( \beta \) and \( \gamma \) 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 \( \text{treeset}(q, \gamma \cup \{c\}) \). Then we have,
\( \beta \cap \gamma = \emptyset \Rightarrow (t_{path}(p, q, c, \beta, \gamma) \ast \text{treeset}(q, \gamma \cup \{c\}) \Rightarrow \text{treeset}(p, \beta \cup \gamma \cup \{c\})) \)

\[ \square \]

**Proof.** For the cases when \( p = q \), we have:

\[
(\beta \cap \gamma = \emptyset) \land t_{path}(p, q, c, \beta, \gamma) \ast \text{subtree}(q, \gamma \cup \{c\})
= (\beta \cap \gamma = \emptyset) \land (c \notin \beta \cup \gamma) \land p = q \land \beta = \emptyset \land \text{emp} \ast \text{subtree}(q, \gamma \cup \{c\})
= \text{true} \land p = q \land c \notin \gamma \land \beta = \emptyset \land \text{subtree}(q, \gamma \cup \{c\})
= \text{subtree}(p, \beta \cup \gamma \cup \{c\})
\]

Now we suppose that \( p \neq q \), and do the induction on the length \( k \) of the path \( (p, p_1, \ldots , p_n, q) \) 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

\[
(\beta \cap \gamma = \emptyset) \land t_{path}(p, q, c, \beta, \gamma) \ast \text{subtree}(q, \gamma \cup \{c\})
= \exists l, r, v, \beta_1, \beta_2, (\beta \cap \gamma = \emptyset) \land (c \notin \beta \cup \gamma) \land (\beta = \beta_1 \cup \{v\} \cup \beta_2) \land \text{bnode}(p, l, v, r) \ast
\]

\[
(t_{path}(\beta_1 \cup \gamma \cup \{c\}, v) \land \text{lower}(\beta_2, v) \land l = q \land t_{path}(l, q, c, \beta_1, \gamma) \ast \text{subtree}(r, \beta_2)) \lor
\]

\[
(t_{path}(\beta_1, v) \land \text{lower}(\beta_2 \cup \gamma \cup \{c\}, v) \land r = q \land t_{path}(r, q, c, \beta_2, \gamma) \ast \text{subtree}(l, \beta_1)) \ast
\]

\[
\text{subtree}(q, \gamma \cup \{c\})
\]

Here we have two sub-cases:

**Case** \( v > c \) :

\[
\exists r, v, \beta_1, \beta_2, (\beta \cap \gamma = \emptyset) \land (c \notin \beta \cup \gamma) \land (\beta = \beta_1 \cup \{v\} \cup \beta_2) \land (\beta_1 = \emptyset) \land
\]

\[
t_{path}(q, q, c, -\gamma) \ast \text{subtree}(q, \gamma \cup \{c\})
\]

**Case** \( v < c \) :

\[
\exists r, v, \beta_1, \beta_2, (\beta \cap \gamma = \emptyset) \land (c \notin \beta \cup \gamma) \land (\beta = \beta_1 \cup \{v\} \cup \beta_2) \land (\beta_1 = \emptyset) \land
\]

\[
t_{path}(q, q, c, -\gamma) \ast \text{subtree}(q, \gamma \cup \{c\})
\]

\[
= \exists r, v, \beta_1, \beta_2, (\beta \cap \gamma = \emptyset) \land (c \notin \beta \cup \gamma) \land (\beta = \beta_1 \cup \{v\} \cup \beta_2) \land (\beta_1 = \emptyset) \land
\]

\[
t_{path}(q, q, c, -\gamma) \ast \text{subtree}(q, \gamma \cup \{c\})
\]

\[
= \text{subtree}(p, \beta \cup \gamma \cup \{c\})
\]

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

\[
(\gamma' = \gamma \cup \beta_n) \land (\beta = \beta' \cup \beta_n)
\]

\[
(\beta' \cap \gamma' = \emptyset) \land t_{path}(p, p_n, c, \beta', \gamma') \ast \text{subtree}(p_n, \gamma', \{c\}) \Rightarrow \text{subtree}(p, \beta' \cup \gamma' \cup \{c\})
\]

\[
(\beta \cap \gamma = \emptyset) \land (\gamma' = \gamma \cup \beta_n) \land (\beta = \beta' \cup \beta_n) \Rightarrow (\beta' \cap \gamma' = \emptyset)
\]

\[
(\beta \cap \gamma = \emptyset) \land (\beta = \beta' \cup \beta_n) \Rightarrow (\beta_n \cap \gamma = \emptyset)
\]
Thus,

\[(\beta \cap \gamma = \emptyset) \land \text{tpath}(p, q, c, \beta, \gamma) \land \text{subtree}(q, \gamma \cup \{c\})\]

\[\Rightarrow (\beta \cap \gamma = \emptyset) \land \text{tpath}(p, p_n, c, \beta', \gamma') \land \text{tpath}(p_n, q, c, \beta_n, \gamma) \land \text{subtree}(q, \gamma \cup \{c\})\]

\[= (\beta \cap \gamma = \emptyset) \land \left((\beta_n = \{v_n\} \cup \beta_1 \cup \beta_2) \land \text{upper}(\beta_1, v_n) \land \text{lower}(\beta_2, v_n) \land \text{tpath}(p_n, q, c, \beta_n, \gamma) \land \text{subtree}(l, \beta_1) \lor (v_n > c \land \text{tpath}(l, q, c, \beta_1, \gamma) \lor \text{subtree}(r, \beta_2))) \land \text{subtree}(q, \gamma \cup \{c\})\right)\]

\[= (\beta_n \cap \gamma = \emptyset) \land ((\beta_n = \{v_n\} \cup \beta_1 \cup \beta_2) \land \text{tpath}(p_n, q, c, \beta', \gamma') \land \text{subtree}(p_n, \beta_n \cup \gamma \cup \{c\})\]

\[= (\beta' \cap \gamma' = \emptyset) \land \text{tpath}(p_n, q, c, \beta', \gamma') \land \text{subtree}(p_n, \gamma' \cup \{c\})\]

\[= \text{subtree}(p, \beta \cup \gamma \cup \{c\})\]

By the induction on the length of the trace, we have the conclusion. \(\square\)

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:**

\{
\text{raw(this, BNode)}
\}

\this.val = v; \this.left = null;
\this.right = null;
\{this.val \mapsto v \times this.left \mapsto \text{null*} this.right \mapsto \text{null}\}
\{\text{bnode(this, null, v, null)}\}

**Proving TreeSet.TreeSet:**

\{
\text{raw(this, TreeSet)}
\}

\this.rt = null;
\{this.rt \mapsto \text{null * emp}\}
\{this.rt \mapsto \text{null * subtree(rnnull, 0)}\}
\{set(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.

\[
\{ b = rfalse \land p = rnull \land set(this, \alpha) \} \\
p = this.rt; \\
\{ \exists p \cdot b = rfalse \land p = r_p \land this.rt \leftarrow r_p \ast subtree(r_p, \alpha) \} \\
\text{if } (p == \text{null}) \\
\{ b = rfalse \land p = rnull \land \alpha = \emptyset \land this.rt \leftarrow rnull \ast \text{emp} \} \\
\{ b = rfalse \land p = rnull \land \alpha = \emptyset \} \\
b = true; \\
\{ b = \text{true} \land \alpha = \emptyset \} \\
\text{else} \\
\{ \exists p, r_1, r_2, \beta_1, \beta_2 \cdot (\alpha = \beta_1 \cup \{ v \} \cup \beta_2) \land \text{upper}(\beta_1, v) \land \text{lower}(\beta_2, v) \land \\
b = rfalse \land p = r_p \land r_p \neq rnull \land this.rt \leftarrow r_p \ast \text{subtree}(r_p, \beta_1) \ast \text{subtree}(r_2, \beta_2) \} \\
b = \text{false}; \\
\{ b = rfalse \land p \neq rnull \land \alpha \neq \emptyset \} \\
\{ b = rfalse \land \alpha \neq \emptyset \} \\
\text{return } b; \\
\{ \text{set}(this, \alpha) \land ((\alpha = \emptyset \land \text{res} = \text{true}) \lor (\alpha \neq \emptyset \land \text{res} = \text{false})) \} \\
\{ \text{set}(this, \alpha) \land \text{res} = (\alpha = \emptyset) \} \\
\]

• TreeSet.add with its specification.

\[
\{ p = rnull \land q = rnull \land t = rnull \land v = 0 \land set(this, \alpha) \} \\
v = 0; \\
\{ p = rnull \land q = rnull \land v = 0 \land set(this, \alpha) \} \\
p = this.rt; q = \text{null}; \\
\{ \exists p, r_q \cdot p = r_p \land q = r_q \land r_q = rnull \land v = 0 \land this.rt \leftarrow r_p \ast subtree(r_p, \alpha) \} \\
\text{while } (p != \text{null}) \\
\{ \exists p, r_q \cdot p = r_p \land q = r_q \land (this.rt \leftarrow r_p \land r_q = rnull \land subtree(r_p, \alpha)) \} \\
(\exists \beta, \gamma, v, \gamma_1, \gamma_2 \cdot r_p \neq rnull \land (\alpha = \beta \cup \gamma) \land (\beta \cap \gamma = \emptyset) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{ v \}) \land \\
\text{upper}(\gamma_1, v) \land \text{lower}(\gamma_2, v) \land (tpath(this.rt, r_q, c, \beta, \gamma) \ast \text{bnode}(r_q, l, v, r) \ast \\
\text{subtree}(l, \gamma_1) \ast \text{subtree}(r, \gamma_2))) \land \\
((c < v \land l = r_p) \lor (c > v \land r = r_p) \lor (c = v \land r_p = \text{null})) \} \\
\{ q = p; \\
\exists p, r_q, v, v', \beta, \beta', \gamma, \gamma', \gamma_1, \gamma_2, \gamma_2' \cdot p = r_p \land q = r_q \land r_q = r_p \land r_q \neq rnull \land \\
r_p \neq rnull \land (\alpha = \beta' \cup \gamma') \land (\beta' \cap \gamma' = \emptyset) \land (\gamma' = \gamma_1' \cup \gamma_2' \cup \{ v' \}) \land \\
(((\beta' = \beta \cup \{ v \} \cup \gamma_1) \land (\gamma' = \gamma_2) \land (v > c)) \lor ((\beta' = \beta \cup \{ v \} \cup \gamma_2) \land (\gamma' = \gamma_1) \land (v < c)) \land \\
\text{upper}(\gamma_1, v) \land \text{lower}(\gamma_2, v) \land tpath(this.rt, r_q, c, \beta', \gamma') \ast \text{bnode}(r_q, l', v', r') \ast \\
\text{subtree}(l', \gamma_1') \ast \text{subtree}(r', \gamma_2')) \} \\
\{ v = q.val; \\
\exists p, r_q, v, \beta', \gamma', \gamma_1', \gamma_2' \cdot p = r_p \land q = r_q \land r_q = r_p \land r_q \neq rnull \land r_p \neq rnull \land \\
(\alpha = \beta' \cup \gamma') \land (\beta' \cap \gamma' = \emptyset) \land (\gamma' = \gamma_1' \cup \gamma_2' \cup \{ v \}) \land \text{upper}(\gamma_1', v) \land \\
\text{lower}(\gamma_2', v) \land tpath(this.rt, r_q, c, \beta', \gamma') \ast \text{bnode}(r_q, l', v, r') \ast \\
\text{subtree}(l', \gamma_1') \ast \text{subtree}(r', \gamma_2') \} \\
\]

if \((v == c)\) 
\[
\{ \exists r_q, r_p, \beta, \gamma, v, \gamma_1, \gamma_2 : p = r_p \land q = r_q \land r_p = r_q \land r_q \neq \text{null} \land c = v \land \\
(\alpha = \beta \cup \gamma) \land (\beta \cap \gamma = 0) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \land \text{upper}(\gamma_2, v) \land \\
\text{lower}(\gamma_2, v) \land \text{tpath}(\text{this}.rt, r_q, c, \beta, \gamma) * \text{bnode}(r_q, l, v, r) * \\
\text{subtree}(l, \gamma_1) * \text{subtree}(r, \gamma_2) \}\n\]
\(p = \text{null};\) 
\[
\{ \exists r_q, \beta, \gamma, v, \gamma_1, \gamma_2 : q = r_q \land r_q \neq \text{null} \land p = \text{null} \land (\alpha = \beta \cup \gamma) \land (\beta \cap \gamma = 0) \land \\
(\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \land \text{upper}(\gamma_1, v) \land \text{lower}(\gamma_2, v) \land v = v \land \\
\text{tpath}(\text{this}.rt, r_q, c, \beta, \gamma) * \text{bnode}(r_q, l, v, r) * \text{subtree}(l, \gamma_1) * \text{subtree}(r, \gamma_2) \}\n\]
\(\{ v : q \neq \text{null} \land p = \text{null} \land c = v \land v \in \alpha \land \text{subtree}(\text{this}.rt, \alpha) \}\) 
if \((v < c)\) 
\[
\{ \exists r_q, r_p, \beta, \gamma, v, \gamma_1, \gamma_2 : p = r_p \land q = r_q \land r_p = r_q \land r_q \neq \text{null} \land r_p \neq \text{null} \land c < v \land \\
(\alpha = \beta \cup \gamma) \land (\beta \cap \gamma = 0) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \land \text{upper}(\gamma_1, v) \land \text{lower}(\gamma_2, v) \land \\
\text{tpath}(\text{this}.rt, r_q, c, \beta, \gamma) * \text{bnode}(r_q, l, v, r) * \text{subtree}(l, \gamma_1) * \text{subtree}(r, \gamma_2) \}\n\]
\(p = q:\text{right};\) 
\[
\{ \exists r_q, r_p, \beta, \gamma, v, \gamma_1, \gamma_2 : q = r_q \land r_q \neq \text{null} \land c > v \land (\alpha = \beta \cup \gamma) \land \\
(\beta \cap \gamma = 0) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \land \text{upper}(\gamma_1, v) \land \text{lower}(\gamma_2, v) \land \\
\text{tpath}(\text{this}.rt, r_q, c, \beta, \gamma) * \text{bnode}(r_q, l, v, r) * \text{subtree}(l, \gamma_1) * \text{subtree}(r, \gamma_2) \}\n\]
\(else\) 
\[
\{ \exists r_q, r_p, \beta, \gamma, v, \gamma_1, \gamma_2 : p = r_p \land q = r_q \land r_p = r_q \land r_q \neq \text{null} \land r_p \neq \text{null} \land c < v \land \\
(\alpha = \beta \cup \gamma) \land (\beta \cap \gamma = 0) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \land \text{upper}(\gamma_1, v) \land \text{lower}(\gamma_2, v) \land \\
\text{tpath}(\text{this}.rt, r_q, c, \beta, \gamma) * \text{bnode}(r_q, l, v, r) * \text{subtree}(l, \gamma_1) * \text{subtree}(r, \gamma_2) \}\n\]
\(p = q:\text{left};\) 
\[
\{ \exists r_q, r_p, \beta, \gamma, v, \gamma_1, \gamma_2 : p = r_p \land q = r_q \land r_q \neq \text{null} \land c < v \land (\alpha = \beta \cup \gamma) \land \\
(\beta \cap \gamma = 0) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{v\}) \land \text{upper}(\gamma_1, v) \land \text{lower}(\gamma_2, v) \land \\
\text{tpath}(\text{this}.rt, r_q, c, \beta, \gamma) * \text{bnode}(r_q, l, v, r) * \text{subtree}(l, \gamma_1) * \text{subtree}(r, \gamma_2) \}\n\]
\(\}
\]
else if (v < c)
    \{∃_t, r_q, v, β, γ, \gamma_1, \gamma_2 \cdot q = r_q \land t = r_t \land r_q \neq \text{null} \land (α = \beta \cup γ) \land (β \cap γ = 0) \land (γ = \gamma_1 \cup \gamma_2 \cup \{v\}) \land (γ_2 = 0) \land (\text{upper(}γ_1, v) \land v < c \land \text{tpath(this.rt, r_q, c, β, γ)*)
        \text{bnode}(r_q, l, v, rnull) * \text{subtree}(l, \gamma_1) * \text{bnode}(r_t, rnull, c, rnull)}\}
q.right = t;
\{∃_t, r_q, v, β, γ, \gamma_1 \cdot q = r_q \land t = r_t \land c \notin α \land r_q \neq \text{null} \land (α = \beta \cup γ) \land (β \cap γ = 0) \land (γ = \gamma_1 \cup \{v\}) \land (\text{upper(}γ_1, v) \land \text{lower(}\{c\}, v) \land \text{tpath(this.rt, r_q, c, β, γ)*)
        \text{bnode}(r_q, l, v, r_t) * \text{subtree}(l, \gamma_1) * \text{subtree}(r_t, \{c\})\}\}
\{∃_t, r_q, β, γ \cdot q = r_q \land t = r_t \land c \notin α \land r_q \neq \text{null} \land (α = \beta \cup γ) \land (β \cap γ = 0) \land (\text{tpath(this.rt, r_q, c, β, γ)*)
        \text{subtree}(r_q, γ \cup \{c\})\}\}
\{c \notin α \land \text{subtree}(\text{this.rt, α} \cup \{c\})\}\}
\}
\{\text{set(this, α} \cup \{c\}\}\}

- TreeSet.contain with its specification.

\{p = \text{null} \land q = \text{null} \land b = \text{rfalse} \land v = 0 \land \text{set(this, α)}\}
\{p = \text{this.rt}; q = \text{null}; b = \text{false}; v = 0;\}
\{∃_p, r_q, r_b \cdot p = r_p \land q = r_q \land b = r_b \land r_q = \text{rnull}\}
r_b = \text{rfalse} \land v = 0 \land \text{this.rt} \rightarrow r_p * \text{subtree}(r_p, α)\}
while (p != \text{null} \land b \text{true})\}
\{∃_p, r_p, r_q, v, β, γ, \gamma_1, \gamma_2 \cdot r_p = r_p \land v = v \land (\text{this.rt} \rightarrow r_p \land r_q = \text{rnull} \land r_b = \text{rfalse} \land \text{subtree}(r_p, α)) \lor (β \cap γ = 0) \land (γ = \gamma_1 \cup \gamma_2 \cup \{v\}) \land (\text{upper(}γ_1, v) \land \text{lower(}γ_2, v) \land r_q \neq \text{null} \land (\text{tpath(this.rt, r_q, c, β, γ)*)
        \text{bnode}(r_q, l, v, r) * \text{subtree}(l, \gamma_1) * \text{subtree}(r_q, γ \cup \{c\})\}\}
\{c \notin α \land \text{subtree}(\text{this.rt, α} \cup \{c\})\}\}
\}
\{q = p;\}
\{∃_p, r_p, r_q, v, β', β'', γ', γ'' \cdot r_q = r_q \land p = r_p \land q = r_q \land r_q = r_p \land
    r_q \neq \text{null} \land r_b = \text{rfalse} \land (α = β' \cup γ) \land (β' \cap γ = 0) \land
    ((β' = β' \cup v) \land (γ' = γ_2) \land (v > c)) \lor ((β' = β' \cup \{v\} \land γ_2) \land (γ' = γ_1) \land (v < c))) \land
    \text{(r''q = \text{r''q} \cup \{v\}) \land \text{upper(}γ', v' \land \text{lower(}γ''v', v') \land \text{tpath(this.rt, r_q, c, β', γ'*)
        \text{bnode}(r_p, l', r', r''q) * \text{subtree}(l', γ'_1) \ast \text{subtree}(r', γ''_2)\}\}
    v = q.val;\}
\{∃_p, r_p, r_q, v, β', γ''_1, γ''_2 \cdot r_q = r_q \land p = r_p \land q = r_q \land r_q = r_p \land r_q \neq \text{null} \land
    r_q \neq \text{null} \land r_b = \text{rfalse} \land (α = β'' \cup γ') \land (β'' \cap γ' = 0) \land (γ' = γ'_1 \cup γ''_2 \cup \{v\}) \land
    \text{upper(}γ', v) \land \text{lower(}γ''v', v') \land \text{tpath(this.rt, r_q, c, β', γ'*)
        \text{bnode}(r_q, l', v, r''q) * \text{subtree}(l', γ'_1) \ast \text{subtree}(r', γ''_2)\}\}
if \((v == c)\)

\[
\{ \\
\exists r, r_p, r_q, \beta, \gamma, v, r_{\gamma_1}, r_{\gamma_2} : b = r_b \land p = r_p \land q = r_q \land r_{\gamma_p} = r_q \land r_r \neq \text{null} \land \\
\quad r_b = \text{rfalse} \land (\alpha = \beta \lor \gamma) \land (\beta \land \gamma = \emptyset) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{ v \}) \land \text{upper}(\gamma_1, v) \land \\
\quad \text{lower}(\gamma_2, v) \land (\text{path}(\text{this}.rt, r_q, c, \beta, \gamma) \land \text{bnode}(r_q, l, v, r) \land \text{subtree}(l, 1)) \land \\
\quad \text{subtree}(r_{\gamma_2}) \land c = v \}
\]

\(b = \text{true};\)

\[
\{ \\
\exists r, r_p, r_q, \beta, \gamma, v, r_{\gamma_1}, r_{\gamma_2} : b = r_b \land p = r_p \land q = r_q \land c = v \land r_r \neq \text{null} \land r_{\gamma_p} = r_q \land \\
\quad r_b = \text{rtrue} \land \text{subtree}(\text{this}.rt, \alpha) \}
\]

if \((v < c)\)

\[
\{ \\
\exists r, r_p, r_q, \beta, \gamma, v, r_{\gamma_1}, r_{\gamma_2} : b = r_b \land p = r_p \land q = r_q \land r_{\gamma_p} = r_q \land r_r \neq \text{null} \land \\
\quad r_b = \text{rfalse} \land (\alpha = \beta \lor \gamma) \land (\beta \land \gamma = \emptyset) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{ v \}) \land \text{upper}(\gamma_1, v) \land \\
\quad \text{lower}(\gamma_2, v) \land (\text{path}(\text{this}.rt, r_q, c, \beta, \gamma) \land \text{bnode}(r_q, l, v, r) \land \text{subtree}(l, 1)) \land \\
\quad \text{subtree}(r_{\gamma_2}) \land c < v \}
\]

\(p = q . \text{right};\)

\[
\{ \\
\exists r, r_p, r_q, \beta, \gamma, v, r_{\gamma_1}, r_{\gamma_2} : b = r_b \land p = r_p \land q = r_q \land r_{\gamma_p} \neq \text{null} \land r_b = \text{rfalse} \land \\
\quad c < v \land (\alpha = \beta \lor \gamma) \land (\beta \land \gamma = \emptyset) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{ v \}) \land \text{upper}(\gamma_1, v) \land \\
\quad \text{lower}(\gamma_2, v) \land (\text{path}(\text{this}.rt, r_q, c, \beta, \gamma) \land \text{bnode}(r_q, l, v, r) \land \text{subtree}(l, 1)) \land \\
\quad \text{subtree}(r_{\gamma_2}) \}
\]

{\(\exists r, r_b, v, \gamma_1, \gamma_2 : q = r_q \land b = r_b \land (r_q = \text{null} \land \alpha = \emptyset \land r_b = \text{rfalse}) \lor \\
\quad (r_q \neq \text{null} \land (\alpha = \beta \lor \gamma) \land (\beta \land \gamma = \emptyset) \land (\gamma = \gamma_1 \cup \gamma_2 \cup \{ v \}) \land \text{upper}(\gamma_1, v) \land \\
\quad \text{lower}(\gamma_2, v) \land (\text{path}(\text{this}.rt, r_q, c, \beta, \gamma) \land \text{bnode}(r_q, l, v, r) \land \text{subtree}(l, 1)) \land \\
\quad \text{subtree}(r_{\gamma_2}) \land (v = c \land r_b = \text{rtrue}) \lor (v \notin \alpha \land r_b = \text{rfalse} \lor \\
\quad (v < c \land r = \text{null} \land \gamma_2 = \emptyset) \lor (v > c \land l = \text{null} \land \gamma_1 = \emptyset))))) \}

\{\exists r, b = r_b \land ((\alpha = \emptyset \land r_b = \text{rfalse}) \lor (\alpha \neq \emptyset \land \{(c \notin \alpha \land r_b = \text{rfalse}) \lor \\
\quad (c \in \alpha \land r_b = \text{rtrue}) \land \text{subtree}(\text{this}.rt, \alpha))))) \}

return \(b;\)

\{\{(c \notin \alpha \land \text{res} = \text{rfalse}) \lor (c \in \alpha \land \text{res} = \text{rtrue}) \} \land \text{set}(\text{this}, \alpha) \}

\{\text{set}(\text{this}, \alpha) \land \text{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.
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**

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.