Formal Verification of ‘Programming to Interfaces’ Programs *

ZHAO Jianhua**, LI Xuandong

State Key Laboratory of Novel Software Technology
Dept. of Computer Sci. and Tech. Nanjing University
Nanjing, Jiangsu, P.R.China 210093
{zhaojh,lxd}@nju.edu.cn

Abstract. This paper presents a formal approach to specify and verify object-oriented programs written in the ‘programming to interfaces’ paradigm. Besides the methods to be invoked by its clients, an interface also declares a set of abstract function/predicate symbols, together with a set of constraints on these symbols. For each method declared in this interface, a specification template is given using these abstract symbols. A class implementing this interface can give its own definitions to the abstract symbols, as long as all the constraints are satisfied. This class implements all the methods declared in the interface such that the method specification templates declared in the interface are satisfied w.r.t. the definitions of the abstract function symbols in this class. Based on the constraints on the abstract symbols, the client code using interfaces can be specified and verified precisely without knowing what classes implement these interfaces. Given more information about the implementing classes, the specifications of the client code can be specialized into more precise ones without re-verifying the client code.
Several commonly used interfaces and their implementations (including Iterator, Observer, Comparable, and Comparator) are used to demonstrate that the approach in this paper is both precise and flexible.

1 Introduction

One of the important programming paradigms of OO programming is ‘programming to interfaces’. Programmers can use an interface without knowing the details of its implementations. This programming paradigm decouples the code using the interfaces and the implementations of these interfaces. It also makes programs more flexible, because programmers can make client code using the interface fulfill different functional features using different implementations of an interface, without modifying the client code. So this paradigm is widely used in software developments, and supported by many modern OO programming languages. In Java, interface is an important concept used to support this paradigm. In C++, abstract base classes play the similar role as interface

* This paper is supported by the Chinese National 863 Project, NO.2011AA010103
** The corresponding author.
does in Java. There are already a number of research works on how to deal with inheritance and method overriding. However, these techniques are not powerful enough to take the full advantage of the 'programming to interfaces' paradigm.

Now we take the interface `java.lang.Comparable` in the standard package of Java as an example to show why these formal techniques are not powerful enough. The interface `Comparable` has one method `compareTo`, which compares an object with the `this` object. A class implementing this interface must define this method. According to the standard documentation of Java, the implementor must ensure that the following formulas hold for any `x, y, z`.

\[
\begin{align*}
- \text{sgn}(x \rightarrow \text{compareTo}(y)) &= -\text{sgn}(y \rightarrow \text{compareTo}(x)); \\
- (x \rightarrow \text{compareTo}(y) > 0 \land y \rightarrow \text{compareTo}(z) > 0) &\Rightarrow x \rightarrow \text{compareTo}(z) > 0; \\
- (x \rightarrow \text{compareTo}(y) = 0) &\Rightarrow (\text{sgn}(x \rightarrow \text{compareTo}(z)) = \text{sgn}(y \rightarrow \text{compareTo}(z))).
\end{align*}
\]

Here `sgn` is a function which yields -1, 0 and +1 respectively when the parameter is less than, equal to or greater than 0.

The above requirements mean that the method `compareTo` induces a total order over the objects. The following code returns the 'smaller' object of the two parameters, given that the two parameters are of the same class implementing the interface `Comparable`.

```java
Comparable TheSmallerOne(Comparable o1; Comparable o2)
{
    if (o1 \rightarrow \text{compareTo}(o2) > 0) return o2; else return o1;
}
```

Different implementations of the method `compareTo` give different meanings to the word 'smaller'.

To support the ‘programming to interfaces’ paradigm, the client code using interfaces must be specified and verified under the open-world assumption, i.e. without knowing what classes implement the interfaces. This is critical because programmers should be able to add a new implementation to an interface without having to re-verify the client code using this interface. Further more, the specification of the client code must be precise and flexible enough, such that programmers can conclude that their implementation of the interface makes the client code fulfill the functional features as expected. For example, suppose that a class `Point` implements the interface `Comparable`. A `Point` object represents a point in the X-Y plate. The method `compareTo` defined in `Point` compares two points by their distances to the original point. The programmer should be able to conclude that the above method returns the point closer to the original point, if the method is invoked using two points as the real parameters.

In the programming language C++, interfaces are written as abstract classes, and they are implemented by their subclasses. People can use the `inheritance` facility to give different implementations to an abstract class. To avoid re-verification of the client code of the abstract classes, researchers deal with `inheritance` based on the Liskov Substitution Principle (LSP) sub-typing rule. For a method overridden in a subclass, the overriding method should have a behavioral sub-typing relation with the overridden one. Roughly speaking, the overriding method should have a weaker precondition and a stronger post-condition.
However, this approach can not be applied in the ‘programming to interfaces’ paradigm because of two reasons.

– An interface (or an abstract class) declares no member variable. So people can not specify how the member variables are accessed and affected by a method when an interface is designed. Specifying a method based only on the parameters and return values is not precise enough. For example, the return value of `compareTo` depends on the member variables of both the `this` object and the real parameter.

– The behavioral sub-typing is too restrictive for the ‘programming to interfaces’ paradigm. In many cases, different implementations of an interface SHOULD behave differently such that the client code using the interface may have different functional features as expected. So even if precise specifications of the interface methods are given somehow, the behavioral sub-typing relation may hurt the flexibility of the paradigm. For example, different implementations of the method `compareTo` should make the method `TheSmallerOne` choose the ‘smaller’ object differently.

In this paper, an approach is presented to specify and verify programs written in the ‘programming to interfaces’ paradigm. The main contributions of this paper are as follows.

– **Abstract but precise specifications of interface methods.** The interface methods are specified through a set of function/predicate symbols and their constraints declared in the interface. These symbols are abstract in the interface, and to be defined in the classes implementing this interface. Because of the constraints, the specification templates given in an interface is precise enough to reason about the client code using this interface.

– **A flexible implementation relation between interfaces and classes.** To implement an interface, a class must define all the function symbols and implement all the methods declared in the interface. The only two restrictions are (1) the function/predicate symbol definitions must satisfy the constraints declared in the interface; (2) the method implementations should satisfy the specification templates declared in the interface w.r.t. the symbol definitions in this class. When the idea in this paper is applied to deal with class inheritance, the LSP sub-typing rule is a special case of our rule.

– **Verifications of the client code under the open-world assumption.** No knowledge about the implementing classes is needed during the verification of the client code of interfaces. Furthermore, when more information about the runtime classes is given, the specifications of the client code can be specialized to more precise ones.

The rest part of this paper is organized as follows. Section 2 gives a brief introduction on the small language used in this paper. The syntax of interface is described in Section 3. The syntax of class definitions is given in Section 4. This section also discusses how to verify the implementation relation between a class and an interface, especially how to verify a method w.r.t. its specifications. The types and expressions associated with interfaces and classes are
discussed in Section 5. The scope memory rules for these expressions are also
given in this section. In Section 6, the syntax and proof rules of the statements
associated with classes and interfaces are given. The approach to verify a client
code using interfaces is presented in Section 7. Section 8 discusses briefly how
to deal with inheritance using the approach presented in this paper. Section 9
compares the method in this paper with related works and concludes this pa-
paper. In Appendixes, we give more examples of some widely-used interfaces, their
implementations, and the client codes using these interfaces.

2 A brief introduction to the small language

This section gives a brief description about the small OO language used in this
paper. More details about this language will be given later.

A program of the small language consists of a set of interface declarations and
class definitions. An interface can be implemented by one or more classes, while
a class can implement zero or more interfaces. In these interface declarations
and class definitions, code are given together with their specifications.

An interface declares a set of methods that can be invoked by its users and
a set of function (predicate) symbols used in method specifications. All these
methods and function symbols are polymorphic, i.e. the classes implementing
the interface can give their own definitions to them. For a method invocation
e→m(), the method definition given in the runtime class of e is invoked. Similarly,
for a term e→f() in specifications, where f is a function symbol declared in an
interface, f also refers to the definition of f in the runtime class of e.

For these polymorphic symbols, class-prefixes are used to specify which defi-
nitions they refer to. For example, C::f refers to the definition of f in the class C.
We use classOf(e) to denote the dynamic class of e, and classOf(e)::f refers to
the definition of f in the dynamic class of e. In interface declarations, we use the
keyword theClass to represent the class implementing this interface. Intuitively
speaking, a sub-expression theClass::f(\(\overline{y}\)) in the declaration of an interface I
has its counterpart C::f(\(\overline{y}\)) in the definition of a class C implementing I.

For each method m declared in an interface I, the templates of the pre-
condition and the post-condition of m are given in I. A template is a formula
containing the keyword theClass. For a class C implementing I, the specifica-
tion of m is derived by substituting theClass with C in the templates.

An interface also declares a set of constraint templates about the function
symbols declared in the interface. In a class C implementing I, a set of constraints
are derived by substituting theClass with C in these constraint templates. The
function symbol definitions in C must satisfy these constraints.

A class definition C defines a set of methods and function symbols. For each
interface I implemented by the class C, all the methods and function symbols
declared in I should be defined in C.

To verify a program, we have the following two kinds of proof obligations.

- A class C defines all the methods and function symbols declared in each
  interface implemented by C. Further more, the function symbol definitions
must satisfy all the constraints derived by substituting \texttt{theClass} with $C$ in the constraint templates in the interfaces.

– Each method defined in a class $C$ should satisfy its specifications. For a method $m$ declared in an interface $I$ implemented by $C$, its specification in $C$ is derived by substituting \texttt{theClass} with $C$ in the specification template of $m$ in $I$.

3 The interface declarations

An interface declares a set of methods that can be invoked by its users. The templates of the pre-conditions and post-conditions of these methods are also given in the interface declaration. An interface also declares a set of function symbols which are used to specify the methods of this interface. A set of constraint templates about the function symbols are also given in the interface. The templates for pre-/post-conditions and constraints are formulas containing the keyword \texttt{theClass}. It will be substituted with the concrete class name in the class definitions implementing this interface.

\begin{figure}[h]
\centering
\begin{verbatim}
ISpec ::= iName '{' fSymDecs Constraints methodDecs '}'
 fSymDecs ::= ε | func ' (' fDec (' ;' fDec)*)
 fDec ::= ( static | attrib | ε ) Type fName '(' fPara ')' Constraints ::= ε | cons ':' Formula (' ;' Formula)*
 methodDecs ::= ε | methods ':' (MSpec)*
 MSpec ::= Type fName '(' fPara ')' pre Formula post Formula ';'
\end{verbatim}
\caption{The grammar of interface specifications}
\end{figure}

The grammar of interface specifications is depicted in Fig. 1. The meta expression $(\ldots)^*$ means that the component in the brackets can repeat for zero or many times. The meta expression $(\ldots)^+$ means that the component in the brackets can repeat for one or many times. An interface declaration consists of three parts: the function symbol declarations, the constraint templates about these function symbols, and the method declarations and specifications. In the grammar, ‘iName’ and ‘fName’ are identifiers, representing interface names and function symbol names respectively. ‘Type’ represents type-expressions, ‘fPara’ represents formal parameter declarations. ‘Formula’ represents boolean-typed expressions. The syntax of Type and Formula are given in [S].

\textbf{Example 1.} An interface \texttt{Comparable} is declared in Fig. 2. This interface is similar to the one given in the standard Java library java.lang. It declares two function symbols \texttt{VALUE} and \texttt{LE}. Three constraint templates about \texttt{VALUE} and \texttt{LE} are given in this interface. This interface declares one method compareTo. The pre-/post-condition templates of \texttt{compareTo} are given using the function symbols \texttt{VALUE} and \texttt{LE}.

\hfill \Box
3.1 Function symbols

The function symbols declared in an interface are used to specify the methods of the interface. For each symbol, the result type, arity, parameter types are given. For each symbol $f$ declared in the interface, the memory scope function symbol, i.e. $\mathcal{M}(f)$, is also implicitly declared in the interface. $\mathcal{M}(f)$ has the same formal parameters as $f$, and the return type of $\mathcal{M}(f)$ is SetOf(Ptr). Based on the rules for memory scopes given in [8], the definition of $\mathcal{M}(f)$ in a class implementing the interface can be constructed syntactically from the definition of $f$ in the same class.

There are two kinds of function symbols: class symbols and object symbols. The object function symbols describe the properties about individual objects, while class symbols describe the properties associated with the class. The symbols declared with the keyword ‘static’ are class symbols, and the rest are object symbols. Given a function symbols $f$, $\mathcal{M}(f)$ and $f$ are in the same kind.

Besides the explicitly declared function symbols, each interface has three additional special object function symbols: SetOf(Ptr) BLOCK(), SetOf(Ptr) pmem(), and bool INV(). Intuitively speaking, BLOCK yields the memory units assigned to the member variables of the object, pmem() yields the private memory owned by the object, and INV() is the invariant of the object. It is required that the invariant of an object $o$, i.e. $o\rightarrow$INV(), holds before/after each method invocation to $o$. For conciseness, the formula this→INV() is usually omitted in the pre-/post-conditions of the methods.

The function symbols declared with the keyword attrib are special object symbols called attribute symbols. This kind of function symbols have no formal parameter. For each attribute symbol $f$ of an interface $I$, there is an implicit constraint template

$$\forall o : \text{theClass}. (o \neq \text{nil} \Rightarrow (o\rightarrow$INV() $\Rightarrow o\rightarrow \mathcal{M}(f)() \subseteq \text{pmem}()$$

Intuitively speaking, attribute symbols should only access the private memory of the object. Both of the function symbols pmem() and INV() are attribute symbols.
In specifications, object function symbols are used in the form $e \rightarrow f(\overline{y})$, where $\overline{y}$ represents a list of suitable real parameters. Usually, we abbreviate $\mathbf{this} \rightarrow f(\overline{y})$ as $f(\overline{y})$. Class function symbols are used in the form $cexp:f(\overline{y})$, where $cexp$ is either a class name, or the keyword $\mathbf{theClass}$, or $\mathbf{classOf}(e)$ for some expression $e$. Usually, we abbreviate $\mathbf{theClass}:f(\overline{y})$ as $f(\overline{y})$.

Example 2. The interface $\mathbf{Comparable}$ in Fig. 2 declares two function symbols: the object symbol $\mathbf{VALUE}$ and the class symbol $\mathbf{LE}$. $\mathbf{VALUE}$ is an attribute symbol. The function symbol $\mathbf{VALUE}$ yields an integer. The class function (predicate) symbol $\mathbf{LE}$ compares two integers.

The object function symbols $\mathbf{INV}$ and $\mathbf{pmem}$ are implicitly declared. The memory scope function symbols $\mathbb{M}(\mathbf{VALUE})$ and $\mathbb{M}(\mathbf{LE})$ are also implicitly declared in $\mathbf{Comparable}$. Because $\mathbf{VALUE}$ is an attribute symbol, there is an implicit constraint template.

$$\forall o: \mathbf{theClass}. (o \neq \mathbf{nil} \Rightarrow (o \rightarrow \mathbf{INV}()) \Rightarrow o \rightarrow \mathbb{M}(\mathbf{VALUE}()) \subseteq \mathbf{pmem}())$$

In the post-condition of the method $\mathbf{compareTo}$, two occurrences of $\mathbf{LE}$ are abbreviations for $\mathbf{theClass}:\mathbf{LE}$; the first and the fourth occurrence of $\mathbf{VALUE}()$ are abbreviations for $\mathbf{this} \rightarrow \mathbf{VALUE}()$. □

3.2 Constraint templates about the function symbols

An interface also declares a set of constraint templates, which are a set of boolean expressions, about the declared function symbols. Each class implementing the interface can give its own definitions to the function symbols declared in the interface. However, these definitions must satisfy all of the constraints derived from these templates by substituting $\mathbf{theClass}$ with the class name. Based on these constraint templates, we can reason about formulas containing these function symbols without knowing the exact definitions to which these symbols refer.

Example 3. The constraint templates declared in the interface $\mathbf{Comparable}$ show that $\mathbf{LE}$ is a total order over integers. Together with the attribute function $\mathbf{VALUE}$, it indirectly casts a total order over the objects.

There is an implicit constraint template associated with each attribute symbol declared in $\mathbf{Comparable}$. Such a constraint template about $\mathbf{VALUE}$ is already given in Example 2. There are also similar constraint templates for $\mathbf{INV}$ and $\mathbf{pmem}$.

For any class implementing $\mathbf{Comparable}$, its definitions to $\mathbf{VALUE}$, $\mathbf{INV}$, $\mathbf{LE}$ and $\mathbf{pmem}$ must satisfy the constraints derived from these templates. □

3.3 Method declarations and specifications

For each method declared in an interface, the method name, the formal parameters, and the return type are given. Besides these information, each method is specified using templates of pre-/post-conditions, which are formulas containing
the keyword theClass. The function symbols declared in the interface can be used in these templates. Suppose that $I$ is an interface, $m$ is a method declared in $I$, we use

$$I::m(x) : \{P\}_{x} \{Q\}$$

to specify that the templates of pre-/post-conditions of $m$ are $P$ and $Q$ respectively. For any class $C$ implementing $I$, the pre-/post-condition of $m$ defined in $C$ are derived by substituting theClass with $C$ in $P$ and $Q$ respectively.

To reason about the relations between the values before/after method invocation, we can use $\overline{e}$ in the post-condition to represent the value of $e$ on the pre-state.

**Example 4.** In Fig. [2] the specification template of compareTo includes the function symbols VALUE and LE. Intuitively speaking, the specification says that the method returns negative, zero, or positive integers respectively when this is less than, equal to, or greater than the parameter $o$. Note that, all the occurrences of LE in the specification template are the abbreviations for theClass::LE.

A class $C$ implementing Comparable can give different definitions to VALUE and LE, denoted as $C$::VALUE and $C$::LE respectively. Note that the two occurrences of LE are abbreviations for theClass::LE, the precondition and post-condition of compareTo in $C$ are respectively

$$\{\rho \land o \neq \text{nil} \land \text{classOf}(o) = C\}$$

and

$$\{\rho \land (C::\text{LE(VALUE()), o->VALUE()) \Rightarrow \text{ret} \leq 0) \land \\
(C::\text{LE(o->VALUE(), VALUE())}) \Rightarrow \text{ret} \geq 0)\}$$

□

### 4 The class definitions

A class definition declares a set of member variables and defines a set of function symbols and methods. For simplicity, we suppose that all the member variables are private (not accessible outside the class definition) and all the function symbols and methods are public (accessible outside the class definition).

A class definition also lists all the interfaces implemented by it. For each interface implemented by this class, the following conditions hold.

- All the function symbols declared in the interface must be defined in this class. All the constraints derived from the constraint templates in the interface should be tautologies.
- All the methods declared in the interface must be defined in the class. The specifications of these methods are derived by substituting theClass by the class name in the corresponding templates in the interface.
A class definition can also define its own function symbols and methods. The preconditions and post-conditions of these methods must be given explicitly. There is one and only one method that has the same name with the class. This method is called the constructor of the class, and is used to create objects of the class. A constructor has no return type.

The grammar of class definitions are depicted in Fig. 3. The meta expression (...)? means that the grammar component in the brackets is optional. In the grammar, ‘cName’, ‘vName’, ‘fName’, ‘mName’ are identifiers respectively representing class names, variable names, function names, and method names. The component ‘iList’ lists the names of the interfaces implemented by this class. The other components are explained in the following subsections.

```
cDef ::= cName impl iList '{' vDecs fDefs mDefs '}'
  vDecs ::= var ':' ( Type vName ';')*  
fDefs ::= ε | func ':' ( funcType Type fName (fPara) ≜ expressions ';' )*  
mDefs ::= method ':' ( mDef )*  
mDef ::= ( Type )? mName '(' fPara ')' ( pre Formula; post Formula; )?  
  ' {' vDecs statementList (return expressions)? ' '  
```

Fig. 3. The syntax of class definitions

Example 5. An example of the class definitions is given in Fig. 4. The class Point implements the interface Comparable. Two member variables x and y are declared in this class. Both of the function symbols VALUE and LE declared in Comparable are defined in Point. This class also defines two attribute functions FldX() and FldY(), which yields the value of the member variables x and y. The method compareTo declared in Comparable is defined in Point. The specification of this method is derived from the corresponding specification template in Comparable by substituting theClass with Point. For conciseness, this method specification is omitted in this class definitions. The constructor of Point is defined in this class. It creates a new object with the member variables x and y set to 0. Three other methods, Set, getX and getY, are defined in this class. The specifications of these methods are explicitly given in this class.

4.1 Member variable declarations

The set of member variables are declared in the part ‘vDecs’. These member variables can only be accessed in the function symbol definitions and the method definitions of this class. In these definitions, a member variable v can be accessed as this→v, or just v for conciseness.

Member variables are private. They are a part of the implementation details of a class, so they should not be exposed to other class definitions. Because
class Point implements Comparable {
    var:
        int x;
        int y;
    funcs:
        attrib int VALUE() = x * x + y * y;
        static bool LE(v1, v2) = v1 ≤ v2;
        attrib int FldX() = x;
        attrib int FldY() = y;
    method:
        Point() {
            x = 0; y = 0;
        }
        void Set(int x1, int y1)
        int getX()
        int getY()
        int compareTo(Comparable* o) {
            int tmp1;
            int tmp2;
            tmp1 = o->getX();
            tmp2 = o->getY();
            return x*x + y*y - tmp1*tmp1 - tmp2*tmp2;
        }
    }
}

Fig. 4. The class Point implementing the interface Comparable

the method specifications will be used to reason about method-invocation statements outside the class definition, member variables can not occur in the method specifications.

Example 6. In the class definition in Fig. 4, two member variables x and y are declared. They can be directly accessed in the definitions of the function symbols and methods in the Point. The keyword this is omitted in these definitions. For example, the definition of VALUE in Point is equivalent to

        attrib int VALUE() = this->x * this->x + this->y * this->y;

These member variables can not appear in the pre-/post-conditions of the methods. Instead, we use the function symbols FldX() and FldY() in the specifications.

4.2 Function symbol definitions

A set of function symbols are defined in the part ‘fDefs’. For each interface implemented by this class, all the function symbols declared in the interface should be defined here. The definition of the mandatory function BLOCK is derived directly from the part ‘vDecs’.

        attrib SetOf(ptr) BLOCK() = {this->&v|v is a member variable in ‘vDecs’}
People can define the mandatory functions \texttt{pmem()} and \texttt{INV()} in the class, or just use the following default definitions.

\begin{verbatim}
attrib SetOf(ptr) pmem() ≜ this→BLOCK()
attrib bool INV() ≜ true
\end{verbatim}

The keyword \texttt{this} can not appear in the definitions of any class function symbols. Member variable access is also forbidden in the definitions of class functions.

Given a function symbol \( f \) defined in the class as

\[ f(\mathcal{F}) \triangleq e \]

the definition of \( \mathcal{M}(f) \) can be construct syntactically from the definition of \( f \) following the rules given in \cite{8}. That is,

\[ \text{Setof(Ptr) } \mathcal{M}(f)(\mathcal{F}) \triangleq \mathcal{M}(e) \]

The function symbol definitions in this class must satisfy all the constraints derived from the constraint templates declared (explicitly or implicitly) in any interface implemented by the class.

\textit{Example 7.} In Fig. 4, the class \texttt{Point} defines four function symbols: \texttt{VALUE}, \texttt{LE}, \texttt{FldX}, and \texttt{FldY}. Among them, \texttt{VALUE} and \texttt{LE} are declared in the interface \texttt{Comparable}. The function \texttt{LE} is a class function symbol, so its definition does not access member variables.

The functions \texttt{pmem} and \texttt{INV} are not explicitly defined in \texttt{Point}, so their definitions are

\begin{verbatim}
attrib SetOf(ptr) pmem() ≜ \{this→x, this→y\}
attrib bool INV() ≜ true
\end{verbatim}

The are three explicit constraint templates about \texttt{LE} in \texttt{Comparable}. Substituting \texttt{theClass} by \texttt{Point}, we have the following three constraints.

\begin{align*}
\forall v: \text{int}.Point::LE(v, v); \\
\forall v_1, v_2: \text{int}.(Point::LE(v_1, v_2) \lor \text{Point::LE}(v_2, v_1)); \\
\forall v_1, v_2, v_3: \text{int}.(Point::LE(v_1, v_2) \land \text{Point::LE}(v_2, v_3) \Rightarrow \text{Point::LE}(v_1, v_3));
\end{align*}

It can be checked that the definition of \texttt{Point::LE} satisfies all these constraints.

The following definition of \( \mathcal{M}(\texttt{VALUE}) \) is derived from the definition of \texttt{VALUE} by the memory scope rule given in \cite{8}.

\begin{verbatim}
attrib SetOf(Ptr) \mathcal{M}(\texttt{VALUE})(\mathcal{F}) ≜ \{\&\textthis→x,\&\textthis→y\};
\end{verbatim}

Because \texttt{VALUE} is an attribute symbol, the definition of \( \mathcal{M}(\texttt{VALUE}) \) must satisfy the following constraint.

\[ \forall o: \text{Point}.(o \neq \text{nil} \Rightarrow (o\rightarrow\text{INV}()) \Rightarrow (o\rightarrow\mathcal{M}(\texttt{VALUE}()) \subseteq o\rightarrow\text{pmem}()))) \]

From the definition of \texttt{Point::pmem}, this constraint is always satisfied. \( \square \)
4.3 Method definitions

A set of methods are defined in the part ‘mDefs’. A class must define one and only one constructor, which is used to create new objects of this class. The general grammar of method definitions is given in Fig. 3. Given a method \( m \) defined in a class \( C \), we use the following formula

\[
C::m : \{P\} - \{Q\}
\]

to express that \( P \) and \( Q \) are respectively the pre-condition and post-condition of the method \( m \) defined in \( C \).

**Constructor definitions.** Constructors are used to create new objects. The constructor of a class has the same name as the class. A constructors do not return a value, so the return type and the ‘return expression’ in the general grammar are absent. As a consequence, the keyword \texttt{ret} does not appear in the post conditions. The object has not been created yet when a constructor is invoked, so the keyword \texttt{this} can not appear in the precondition of a constructor.

**Common method definitions.** For each interface implemented by a class, all the methods declared in the interface should be defined in the class. A class can also define its own methods. If the return type of this method is \texttt{void}, the ‘return expression’ in the general grammar is absent and the keyword \texttt{ret} does not appear in the post-condition.

Let \( m \) be a method declared in an interface \( I \), and \( C \) be a class implementing \( I \). The specification of \( C::m \), i.e. the method defined in \( C \), is derived by substituting \texttt{theClass} with \( C \) from the corresponding template declared in the interface. That is, \( I::m : \{P\} - \{Q\} \) implies \( C::m : \{P[C\texttt{/theClass}]\} - \{Q[C\texttt{/theClass}]\} \). For conciseness, the pre-/post-condition of \( m \) are absent in the class definition in \( C \).

**Example 8.** The class \texttt{Point} in Fig. 4 defines the constructor of \texttt{Point}, which creates a new object with the fields \( x \) and \( y \) set to 0. The method \texttt{compareTo} is declared in the interface \texttt{Comparable}. The specification of \texttt{compareTo} is not explicitly given in the class definition. It is derived by substituting \texttt{theClass} with \texttt{Point} in the template given in \texttt{Comparable}. So the precondition and post-condition are respectively

\[
\{\rho \land o \neq \texttt{nil} \land \texttt{classOf}(o) = \texttt{Point}\}
\]

and

\[
\{\rho \land (\texttt{Point::LE(VALUE(), o\rightarrow VALUE())} \Rightarrow \texttt{ret} \leq 0) \land \\
\{\texttt{Point::LE(o\rightarrow VALUE(), VALUE())} \Rightarrow \texttt{ret} \geq 0}\}
\]

Three other methods are defined in \texttt{Point}: \texttt{getX}, \texttt{getY}, and \texttt{set}. They are used to access the member variables of the objects. \[ \square \]
4.4 Verifying a method w.r.t. its specification

In this subsection, we discuss how to verify a method w.r.t. its specification. For each method \( m \) defined in a class \( C \), if \( m \) is declared in an interface implemented by \( C \), its specification can be derived by a substitution in the corresponding template; Otherwise, its specification is given in \( C \) explicitly.

**Verification of constructor specifications.** Let the constructor of a class \( C \) be defined as

\[
C(x) \ \text{pre} \ P; \ \text{post} \ Q; \ \{vDecs \ \text{statement}\}
\]

An execution of this constructor first allocates a new memory block for the new objects, and then executes the statement. The formula \( P \) holds when statement begins it execution if no local variable in \( vDecs \) occurs in \( P \). The keyword \textit{this} refers to the newly allocated memory block for the class \( C \) when statement begins its execution. Let \( \text{LocMem} \) be the abbreviation for

\[
\{\&v \ | \ v \text{ is a local variable declared in } vDecs\}
\]

Let \textit{ExtraPre\_Constructor} be the abbreviation for

\[
\text{this} \neq \text{nil} \land (\text{this} \to \text{BLOCK}() \cap \mathcal{M}(P) = \emptyset) \land (\text{LocMem} \cap \mathcal{M}(P) = \emptyset)
\]

We have that \textit{ExtraPre\_Constructor} holds when statement begins it execution.

There may be some terms of the form \( e \) in the post-condition \( Q \). The expression \( e \) means the value of \( e \) at the pre-state of statement. During the verification of the method body, \( e \) is equivalent to \( e@1 \), where 1 is the entry program point of the method body. Let \( Q' \) be a formula derived by substituting all the sub-expressions of the form \( e \) with \( e@1 \) in \( Q \). After the execution, the memory units for local variables are de-allocated. The formula \( Q' \) is not affected by the de-allocation if no local variable occurs in it.

The precondition \( P \) and post-condition \( Q \) may contain some assertion variables, which can be substituted with other formulas when the specification of \( C::m \) is used at the invocation place of \( C::m \). However, the memory scopes of these formulas does not contains the memory units allocated for the local variables because these memory units are unreachable at the invocation place. So the memory scopes of \( P \) and \( Q \) is disjoint with \( \text{LocMem} \) if \( P \) and \( Q \) contains no local variable declared in \( vDecs \).

Based on the above discussion, we have the following proof rule.

\[
\begin{array}{c}
\text{CSTOR} \\
\{ P \land \text{ExtraPre\_Constructor} \} \ \text{statement} \ \{Q'\}
\end{array}
\]

\[
C::C(x) : \{P\} \downarrow (Q)
\]

* \( P \) and \( Q \) contains no local variable declared in \( vDecs \)
* \( Q' \) is derived by substituting all sub-expression \( e \) with \( e@1 \) in \( Q \)
Example 9. The symbols FldX and FldY (abbreviations for \textit{this}→FldX and \textit{this}→FldY respectively) in the specification of \texttt{Point::Point} refer to their definitions in \texttt{Point}. It can be proved that the following specification holds.

\[
\{ ρ \land \textit{this} \neq \texttt{nil} \land (\texttt{BLOCK}(\rho) \cap \mathcal{M}(\rho) = \emptyset) \}
\]

\[
x = 0; y = 0;
\]

\[
\{ ρ \land (\texttt{BLOCK}(\rho) \cap \mathcal{M}(\rho) = \emptyset) \land \text{FldX}(\rho) = 0 \land \text{FldY}(\rho) = 0 \}
\]

No local variable is declared in this constructor. From the proof rule CSTOR, we prove the following specification.

\texttt{Point::Point() : \{ρ\} → \{ρ \land (\texttt{BLOCK}(\rho) \cap \mathcal{M}(\rho) = \emptyset) \land \text{FldX}(\rho) = 0 \land \text{FldY}(\rho) = 0 \}}

That is, the definition of \texttt{Point::Point} satisfies its specification. □

Verification of common method specifications. Suppose that the body of a method \textit{m} defined in the class \texttt{C} is \{\texttt{vDecs; sts;}\}, and the precondition and post-condition are \textit{P} and \textit{Q} respectively. When \texttt{sts} begins its execution, \textit{P} holds if it contains no local variable in \texttt{vDecs}; \textit{this} is a non-nil pointer. The memory units for local variables are newly allocated. Let \texttt{LocMem} be the abbreviation for \{\&v | v is a local variable declled in \texttt{vDecs}\}

We know that (\mathcal{M}(\rho) \cap \texttt{LocMem} = \emptyset) also holds when \texttt{sts} begins it execution.

Let \textit{Q}' be a formula derived by substituting all the sub-expressions of the form \(\overline{e}\) with \(e@1\) in \textit{Q}. After the execution of statement, the memory units allocated for local variables are de-allocated and the method returns. If \textit{Q}' holds after the execution of \texttt{sts}, and no local variable occurs in \textit{Q}', \textit{Q} holds after the invocation returns.

Based on the above discussion, we have the following proof rule.

\[
\textbf{METHOD} \quad \{P \land \textit{this} \neq \texttt{nil} \land (\mathcal{M}(\rho) \cap \texttt{LocMem} = \emptyset)\} \quad \texttt{sts} \quad \{Q'\}
\]

\[
C::m(\overline{x}) : \{P\} → \{Q\}
\]

\* \texttt{P} and \texttt{Q} contains no local variable declared in \texttt{vDecs}

\* \textit{Q}' is derived by substituting all sub-expression \(\overline{e}\) with \(e@1\) in \textit{Q}

Example 10. The method \texttt{compareTo} is declared in \texttt{Comparable}. Its pre-/post-conditions are given in Example 8. Let \texttt{compareTo\_Pre} and \texttt{compareTo\_Post} are respectively abbreviations for these two formulas. The specification of \texttt{compareTo} can be written as

\texttt{Point::compareTo(o) : \{compareTo\_Pre\} → \{compareTo\_Post\}}

There is no sub-expressions of the form \(\overline{e}\) in \texttt{compareTo\_Post}. According to the proof rule METHOD, this specification holds if the following specification holds.

\[
\{\text{compareTo\_Pre} \land (\{\&\texttt{tmp1}, \&\texttt{tmp2}\} \cap \mathcal{M}(\rho) = \emptyset)\}
\]

The body of \texttt{Point::compareTo}

\{compareTo\_Post\}

□
5 Types and expressions

In this subsection, we discuss the types, expressions, and statements associated with interfaces and classes.

5.1 Types

The small language used in this paper is slightly different from the one used in \[8\]. It supports \texttt{int}, \texttt{bool}, array types, struct types, pointer types, interface types, and class types.

- An interface $I$ declared in the program is a type. The value of an expression with a static type $I$ is a reference to an object of some class implementing $I$.
- A class $C$ defined in the program is also a type. The value of an expression with type $C$ is a reference to an object of the class $C$.

One memory unit is assigned to each variable declared with a class type or an interface type in the runtime. The member variables of an object is stored in a memory block. The memory layout for objects are same as the memory layout for record types in \[8\]. In memory assignment, a variable with type $I$ or $C$ is treated as a pointer to a record type, and the member variables are treated as fields of the record type. So we can still use the axioms for memory layout of record types in that paper.

If $C$ is a class implementing $I$, $I$ is a super type of $C$. That is, a $C$ object can be assigned to a variable with a static type $I$. Given an expression $e$ with static type $I$, the value of $e$ in the runtime is either \texttt{nil} or refers to an object of some class $C'$ implementing $I$. We call $C'$ the \textit{dynamic class} of $e$, denoted as $\text{classOf}(e)$.

Because inheritance is not supported in the small programming language in this paper, given an expression $e$ with static type $C$, the value of $e$ in the runtime is either \texttt{nil}, or refers to an object of $C$.

5.2 Expressions associated with interfaces and classes.

There are five kinds of expressions associated with interfaces used in programs.

- \texttt{this}. It is a keyword used in interface declarations and class definitions. It refers to the current object being manipulated. This keyword can not appear in the constraints, constraints templates, and the definitions of class function symbols. The static type of \texttt{this} is $I$ (or $C$) when it appears in the declaration of an interface $I$ (or the definition of a class $C$).
- \textbf{Member variables}. In our language, the member variables declared in classes are always private. A member variables $v$ in a class $C$ can only occur in the method bodies and object-function symbol definitions in $C$, in the form $\texttt{this} \rightarrow v$. Usually, this expression is abbreviated as $v$ for conciseness. The static type of this expression is just the declared type of $v$. This expression has a left-value, i.e. a value can be assigned to $\texttt{this} \rightarrow v$. 

– **ret.** It is a keyword used only in the post-conditions of methods. It represents
the return value of the method. The static type of **ret** is the return type of
the method.

– **Object function symbol calls.** Let \( f \) be an object function symbol declared as
\[
T \ f(x)
\]
in an interface \( I \) (or defined in a class \( C \)), \( e \rightarrow f(y) \) is an expression with
static type \( T \) if the following conditions hold.

* The static type of \( e \) is \( I \) (or \( C \) respectively).
* The real parameters \( y \) conform to the formal parameters \( x \) of \( f \).

This expression calls the definition of \( f \) in the class \( getClass(e) \). For concise-
ness, the expression \( \textbf{this} \rightarrow f(y) \) is usually abbreviated as \( f(y) \).

– **Class function symbol calls.** Let \( f \) be a class function symbol declared as
\[
T \ f(x)
\]
in an interface \( I \) (or defined in a class \( C \)), \( cexp::f(y) \) is an expression with
static type \( T \) if the following conditions hold.

* The real parameters \( y \) conform to the formal parameter \( x \) of \( f \);
* The class-prefix \( cexp \) is either a class name, or the keyword \( \textbf{theClass} \), or
\( getClass(e) \) for some expression \( e \) with static type \( I \) (or \( C \) respectively).

If \( cexp \) is a class name, the class must implement \( I \) (or just \( C \)).
This expression calls the definition of \( f \) in the class denoted by \( cexp \). For con-
ciseness, the expression \( \textbf{theClass} :: f(y) \) in an interface declaration is usually
abbreviated as \( f(y) \). In a class definition \( C \), \( C::f(y) \) is usually abbreviated
as \( f(y) \).

The memory scope rules for these expressions are given in Table 1.

| The expressions | The memory scopes |
|-----------------|-------------------|
| a class name    | \( \emptyset \)    |
| theClass        | \( \emptyset \)    |
| getClass(e)     | \( 2R(e) \)        |
| this            | \( \emptyset \)    |
| this\( \rightarrow v \) & this\( \rightarrow v \)  |
| e\( \rightarrow f(y) \) | \( 2R(e) \cup 2R(f) \cup e \rightarrow 2R(f)(y) \) |
| cexp::f(y)      | \( 2R(cexp) \cup 2R(y) \cup cexp::2R(f)(y) \) |

| **Table 1.** The memory scopes of expressions associated with interfaces and classes |

6 **Statements and their proof rules**

In the small language used in this paper, the **alloc** statement is no longer sup-
ported. Instead, people can use the object creation statement to create objects.
Three new kinds of statements is discussed in this subsection: the return statements, the object creation statements, and the method invocation statements.

6.1 The return statements.

The return statement is the last statement of a method definition. The statement 'return exp' first evaluates the value of exp, and then returns this value. So we have the following axiom.

\[
\text{RETURN-ST: } \{Q[\text{exp/ret}] \} \text{ return } \text{exp} \{Q\}
\]

6.2 Object creation statements.

Given a class \( C \), the following statement

\[
e_0 := \text{new } C(\overline{y})
\]

first evaluates the left-value of \( e_0 \), (i.e. \&\( e_0 \)), and then creates a new object using the real parameter \( y \), and finally stores the reference into the memory unit referred by \&\( e_0 \). The left-hand \( e_0 \) must have a left-value and the static type of \( e_0 \) is either \( C \), or an interface \( I \) implemented by \( C \). The real-parameters \( \overline{y} \) conforms to the formal parameters of the constructor of \( C \).

When this statement begins its execution, the precondition of the constructor must be satisfied. After the execution, the formula derived by substituting this and formal parameters with the object reference and real parameters respectively in post-condition of the constructor of \( C \) holds. So we have the following proof rule.

\[
\text{OBJ-CREATION} \quad \text{cName::cName(\overline{\tau}) : \{}P\} \{Q\}
\]

\[
\{i : (\&e_0 \neq \text{nil}) \land P[\overline{\tau}/\overline{\tau}]\}
\]

\[
e_0 = \text{new } \text{cName}(\overline{\tau})
\]

\[
\{j : (\ast(\&e_0@i) \neq \text{nil}) \land Q[\ast(\&e_0@i)/\text{this}][\overline{\tau}@i/\overline{\tau}]\}
\]

6.3 Class method invocation statements.

A method \( m \) defined in a class \( C \) can be invoked as \( e_0 := e_1 \rightarrow m(\overline{y}) \) if the return type of \( m \) is not void; or otherwise invoked as \( e_1 \rightarrow m(\overline{y}) \). It is required that \( e_1 \) is an expression with static type \( C \) and the real parameters \( \overline{y} \) must conform to the formal parameters of \( m \). For the first form, \( e_0 \) must have a left-value and the type of \( e_0 \) is a super type of the return type of \( m \).

The statement \( e_0 := e_1 \rightarrow m(\overline{y}) \) invokes the method \( C::m \), i.e. the method \( m \) defined in the class \( C \). Let \( C::m : \{}P\} \{Q\} \) be the specification about \( C::m \). Before the invocation, \( e_1 \) must refer to an object of \( C \), and \&\( e_0 \) must be a non-nil pointer. Furthermore, the precondition \( P[\overline{\tau}/\overline{\tau}][e_1/\text{this}] \) must hold. After the
invocation, the property \( Q[*(&e_0@i)/\mathit{ret}]_{\mathit{classOf}(e) / \mathit{classOf}(e)}[e_1@i/\mathit{this}] \) holds. So we have the following proof rule for the class method invocations of the first form.

\[
{\text{C-INVOC-1}} \quad C::m(\mathit{I}) : \{P\} \rightarrow \{Q\} \quad \begin{array}{l}
i : (e_0 \neq \text{nil}) \land (e_1 \neq \text{nil}) \land P[\mathit{I}/\mathit{I}][e_1/\mathit{this}] \\
e_0 = e_1 \rightarrow m(\mathit{I}) \\
j : Q[*(&e_0@i)/\mathit{ret}]_{\mathit{classOf}(e) / \mathit{classOf}(e)}[e_1@i/\mathit{this}] \end{array}
\]

Similarly, we have the proof rule for the class method invocations of the second form.

\[
{\text{C-INVOC-2}} \quad C::m(\mathit{I}) : \{P\} \rightarrow \{Q\} \quad e_1 \rightarrow m(\mathit{I}) \quad j : Q[\mathit{I}/\mathit{I}][e_1@i/\mathit{this}] \]

### 6.4 Interface method invocation statements.

A method \( m \) declared in an interface \( I \) can be invoked as \( e_0 := e_1 \rightarrow m(\mathit{I}) \) if the return type of \( m \) is not \texttt{void}; or invoked as \( e_1 \rightarrow m(\mathit{I}) \) otherwise. It is required that \( e_1 \) is an expression with static type \( I \) and the real parameters \( \mathit{I} \) must conform to the formal parameters of \( m \). For the first form, \( e_0 \) must have a left-value and the type of \( e_0 \) is a super type of the return type of \( m \).

For conciseness, we usually use \( e_0 := m(\mathit{I}) \) or \( m(\mathit{I}) \) as abbreviations for \( e_0 := \mathit{this} \rightarrow m(\mathit{I}) \) or \( \mathit{this} \rightarrow m(\mathit{I}) \) respectively.

The statement \( e_0 := e_1 \rightarrow m(\mathit{I}) \) invokes the method \( m \) defined in the class \texttt{classOf}(\( e_1 \)) by the real-parameters \( \mathit{I} \), and store the return value in the memory unit &\( e_0 \). Suppose that \( I::m : \{P\} \rightarrow \{Q\} \) holds, and \( C_1, C_2, \ldots, C_n \) are all the classes implementing \( I \). The dynamic class of \( e_1 \), i.e. \texttt{classOf}(\( e_1 \)), is \( C_k \) for some \( 1 \leq k \leq n \) if \( e_1 \neq \text{nil} \). This statement invokes \( C_k::m \) when the dynamic class of \( e_1 \) is \( C_k \). According to the discussion about method specification in Subsection 4.3.3 the preconditions of the method \( C_k::m \) is \( P[C_k/\texttt{theClass}] \). From the discussion about class invocations, the preconditions of the statement is \( e_1 \neq \text{nil} \) and

\[
(\texttt{classOf}(e_1) = C_1)?P[C_1/\texttt{theClass}][\mathit{I}/\mathit{I}][e_1/\mathit{this}] \\
(\texttt{classOf}(e_1) = C_2)?P[C_2/\texttt{theClass}][\mathit{I}/\mathit{I}][e_1/\mathit{this}] \\
\ldots \ldots \\
(\texttt{classOf}(e_1) = C_n)?P[C_n/\texttt{theClass}][\mathit{I}/\mathit{I}][e_1/\mathit{this}] \\
\text{: false}
\]

The above conditional formula is equivalent to

\[
P[\texttt{classOf}(e_1)/\texttt{theClass}][\mathit{I}/\mathit{I}][e_1/\mathit{this}]\]

Let \( i \) be the program point before this invocation statement, the post-condition of the statement \( e_0 := e_1 \rightarrow m(\mathit{I}) \) is

\[
Q[\texttt{classOf}(e_1@i)/\texttt{theClass}][*(&e_0@i)/\mathit{ret}][\mathit{I}/\mathit{I}][e_1@i/\mathit{this}]\]
So we have the following proof rule.

\[
\text{I-INVC-1} \quad \begin{array}{l}
I::m(\overline{x}) : \{P\}.\{Q\} \\
\{i : (\&e0 \neq \text{nil}) \land (e1 \neq \text{nil}) \land \\
P[\text{classOf}(e1)/\text{theClass}]|\overline{y}/\overline{x}[e1/\text{this}]\}
\end{array}
\]

\[
e0 = e1 \rightarrow m(\overline{y}) \\
\{j : Q[\text{classOf}(e1@i)/\text{theClass}]|\overline{y}@i/\overline{x} [e1@i/\text{this}] [e0@i/\text{ret}]\}
\]

For the interface method invocation statements of the form \(e1 \rightarrow m(\overline{y})\), we can similarly get the following proof rule.

\[
\text{I-INVC-2} \quad \begin{array}{l}
I::m(\overline{x}) : \{P\}.\{Q\} \\
\{i : (e1 \neq \text{nil}) \land P[\text{classOf}(e1)/\text{theClass}]|\overline{y}/\overline{x}[e1/\text{this}]\}
\end{array}
\]

\[
e1 \rightarrow m(\overline{y}) \\
\{j : Q[\text{classOf}(e1@i)/\text{theClass}]|\overline{y}@i/\overline{x} [e1@i/\text{this}]\}
\]

7 Code verification under the open-world assumption

The proof rules I-INVC-1 and I-INVC-2 show that we can have the preconditions and post-conditions of the interface method invocation statements without knowing the exact dynamic class of the receivers. These preconditions and post-conditions usually contain terms of the form \(\text{classOf}(e)\), or \(\text{classOf}(e)::f(\overline{y})\). Next, we will discuss how to deal with such terms.

If the dynamic class of an expression \(e\) is \(C\), we know that for any formula template \(P\), \(P[C/\text{theClass}]\) is equivalent to \(P[\text{classOf}(e)/\text{theClass}]\). Let \(I\) be an interface, \(C_1, C_2, \ldots, C_n\) be all the classes implementing \(I\), and \(\text{constr}\) be a constraint template declared in \(I\). According to the proof obligations of class definitions, \(\text{constr}[C_k/\text{theClass}]\) holds for each \(k (1 \leq k \leq n)\). Given an expression \(e\) with static type \(I\), the dynamic class of \(e\), i.e. \(\text{classOf}(e)\), must be some class implementing \(I\) if \(e\) is not \(\text{nil}\). So the following formula holds and can be used in the code verification.

\[
(e \neq \text{nil}) \Rightarrow \text{constr[\text{classOf}(e)/\text{theClass]}
\]

This formula holds even if some new classes implementing \(I\) are added into the program, because the function symbol definitions in these new classes must also satisfy the constraints.

\textbf{Example 11.} Part of the class \texttt{arrayList} is given in Fig.\texttt{5} An \texttt{arrayList} object stores some \texttt{Comparable} objects in the member variable \texttt{arr}. The specification of the method \texttt{sort} is given in the class definition. If all the elements of \texttt{arr} are not \texttt{nil} and point to objects of the same class, the method \texttt{sort} can sort these objects w.r.t. to the order induced by the method \texttt{compareTo}.
class arrayList{
  var:
  Comparable arr[10];
  funcs:
  SetOf(Ptr) pmem()≜ λx.&arr[x][0..9];
  Comparable get(int i)≜ arr[i];
  bool MemLayOut()≜ ∀i∈(0..9)(∀j∈(0..9)(&arr[j]∉get(i)→pmem()));

  method:
  void Set(Comparable obj, int i)
  Pre  ρ ∧ (M(ρ) ∩ pmem()) = ∅ ∧ 0 ≤ i ∧ i ≤ 9
  Post ρ ∧ (get(i) = obj) ∧ ∀x ∈ (0..9)(x ≠ i ⇒ (get(x) = get(x)))
  { arr[i] = obj; }

  void Sort()
  Pre  ρ ∧ (M(ρ) ∩ pmem()) = ∅ ∧ MemoryLayout()
  ∀i ∈ (0..9)(get(i) ≠ nil) ∧ (∀x ∈ (0..9)(classOf(get(i)) = classOf(get(0))))
  Post ρ ∧ (classOf(get(0)) = classOf(get(0))) ∧
  ∀i ∈ (0..8)(classOf(get(i)) :: LE(get(i)→VALUE(), get(i+1)→VALUE())))
  { int i,j,cR;
    Points *tmp;
    i = 9;
    while(i>0)
    { j = 0;
      while (j<i-1)
      { cR = arr[j]→compareTo(classOf(arr[j]))arr[j+1]);
        if(cR > 0)
        { temp = arr[j]; arr[j] = arr[j+1]; arr[j+1] = temp;
          break;
        } else
          skip;
        j = j+1;
      } i = i-1;
    } }

  //The code using ArraySort
  ArrayList* al = new ArraySort();
  //Add ten Point objects into the object al.
  . . . . .
  . . . . .
  al→Sort();

Fig. 5. The sort algorithm for Comparable objects
Because of the space limitation, we just briefly show how to prove that the following formula is an invariant of the inner while-statement.

\[
\forall x \in (0..j - 1) (\text{classOf}(\text{arr}[0]@1):\text{LE}(\text{arr}[x] \rightarrow \text{VALUE()}, \text{arr}[j] \rightarrow \text{VALUE}()) \land \\
\forall x \in (0..9) (\text{arr}[x] \neq \text{nil}) \land \\
\forall x \in (0..9) (\text{classOf}(\text{arr}[x]) = \text{classOf}(\text{arr}[0]@1))
\] (1)

Now we prove that Formula 1 holds at the end of the loop-body if it holds at the beginning of the loop-body.

Because &cR is not in the memory scope of Formula 1, Formula 1 still holds at the program point after the assignment to cR, i.e. the point before the if-statement. Furthermore, based on the specification of compareTo and the proof rule I-INVC-1, the following formula also holds at the point after the assignment to cR.

\[
(\text{classOf}(\text{arr}[j]):\text{LE}(\text{arr}[j] \rightarrow \text{VAL}(), \text{arr}[j+1] \rightarrow \text{VAL}()) \Rightarrow cR \leq 0) \land \\
(\text{classOf}(\text{arr}[j]):\text{LE}(\text{arr}[j+1] \rightarrow \text{VAL}(), \text{arr}[j] \rightarrow \text{VAL}()) \Rightarrow cR \geq 0)
\] (2)

Let CLS be an abbreviation for classOf(arr[0]@1). Because arr[0]@1 \neq nil holds, substituting theClass with classOf(arr[0]@1) in the constraint templates declared in Comparable, we have

\[
\forall v_1, v_2, v_3 : \text{int}. (\text{CLS}::\text{LE}(v_1, v_2) \land \text{CLS}::\text{LE}(v_2, v_3) \Rightarrow \text{CLS}::\text{LE}(v_1, v_3)) \\
\forall v_1, v_2 : \text{int}. (\text{CLS}::\text{LE}(v_1, v_2) \lor \text{CLS}::\text{LE}(v_2, v_1))
\]

From these constraints, it can be implied from Formula 1 and 2 that

\[
(cR \leq 0)? \\
\forall x \in (0..j + 1) (\text{CLS}::\text{LE}(\text{arr}[x] \rightarrow \text{VAL}(), \text{arr}[j+1] \rightarrow \text{VAL}())) \\
: \forall x \in (0..j + 1) (\text{CLS}::\text{LE}(\text{arr}[x] \rightarrow \text{VAL}(), \text{arr}[j] \rightarrow \text{VAL}()))
\]

Using the weakest-precondition computation algorithm presented in [8], it can be proved that the invariant holds at the end of the loop body.

A piece of client code using this class is given following the definition of ArrayList. An ArrayList object al is first created and then ten Point objects are set into this object. The following condition holds at the point before the statement al→Sort().

\[
\forall x \in (0..9) (\text{al}→\text{get}(i) \neq \text{nil}) \land \text{al}→\text{MemoryLayout()} \land \\
\forall x \in (0..9) (\text{classOf}(\text{al}→\text{get}(i)) = \text{Point})
\]

So after this statement, the following property holds.

\[
\forall i \in (0..8) (\text{Point}::\text{LE}(\text{al}→\text{get}(i) \rightarrow \text{VALUE}(), \text{al}→\text{get}(i+1) \rightarrow \text{VALUE}()))
\]
8 Some discussions on encapsulation and inheritance

8.1 About the pmem and INV functions

The function symbols pmem and INV are important in the approach presented in this paper.

Given an object $o$, the expression $o \rightarrow \text{pmem}()$ represents the private memory owned by this object. For any attribute symbol $\text{att}$ of $C$, the value of $o \rightarrow \text{att}()$ keeps unchanged if no memory unit in $o \rightarrow \text{pmem}()$ is changed. A good programming practice is to make each object manage its own private memory and do not share them with other objects. Such features can be checked through the escape analysis technique[7]. For such an object $o$, we have the following conclusions.

- A method invocation $o \rightarrow m$ may modify the private memory of $o$. The properties about the attributes of $o$ can be derived from the specification of $m$;
- For other statements, the private memory of $o$ keeps unchanged, so all the attribute values of $o$ keep unchanged.

From these conclusions, we can reason about the attribute values without referring to the complicated internal structure of the objects. The private memory of such an object can be managed as a whole during the verification.

The function symbol INV is an attribute one, and denotes the invariant of an object. This invariant must hold before/after each method invocation. If an object does not share its private memory with other objects, it is easy to check that the invariant of this object always holds before/after any method invocation. Subclasses can have stronger invariants than their super-classes. So a subclass can define an overriding method that has a stronger invariant as a part of its precondition, while the client code don’t have to be re-verified.

Note that, an object may be designed to share its private memory with others for some reasons. For example, a Container object in Java shares its private memory with its Iterator objects. In such cases, we may have to treat the invariants carefully. It is important to make sure that a method of an object does not break the invariants of another related object.

8.2 About class inheritance

For simplicity, we only discuss in this paper how to deal with the implementation relation between interfaces and classes. However, the same idea can also be applied to class inheritance.

In an ordinary object-oriented language, the definition of a class $B$ can be viewed from two aspects.

- The interface of $B$, which specifies how the class can be used. Here, the interface means only the set of method signatures of $B$;
- The implementations of $B$, i.e. the implementations of the method signatures of $B$. 
A sub-class $D$ of $B$ inherits (and may extends) the interface of $B$. It overrides (i.e. gives new implementations to) some methods of $B$ and inherits the others. If people want $D$ be a subtype of $B$, i.e. a $D$ object can be used in the places a $B$ object is used, some restrictions on the new implementations are required.

Now we discuss two slightly different cases. The first, a superclass is known to be extended by some subclasses at the design time; The second, a new subclass is added after the superclass has been finished. The second case usually happens in the software maintenance phase.

In the following discussion, it is assumed that all the methods are specified by function symbols defined in their classes. No member variable is referred directly in the specification.

**The case a superclass is designed to be inherited.** In this case, programmers should know how the super class is used. So they can give a set of properties about the function symbols defined in the superclass $B$ such that the following conditions hold.

- The properties hold w.r.t. the function symbol definitions given in $B$.
- These properties are sufficient to verify the client code using the superclass $B$. The symbol definitions in $B$ are not directly used in the verification.

Thus, the signatures of function symbols and methods, together with these properties, compose a complete interface specifications of $B$. Programmers can define the subclass $D$ by implementing this interface specification as described in the previous sections: some function symbols and method are inherited, and the others are overridden. All the constraints (properties) given in $B$ must be satisfied.

It is not necessary to re-verify a method inherited from $B$ if none of the function symbols occurred in its specifications are overridden in $D$. Otherwise, the method must be re-verified w.r.t. the new function symbol definitions in $D$.

Roughly speaking, we can view the class definition of $B$ as an interface specification plus a class definition in this case.

**To design a subclass when the superclass and the client code has been finished.** As the interface specification of the superclass is not given, the definitions of function symbols in the superclass are used directly to verify the client code using this superclass.

To implement a subclass such that it can substitute the superclass in the client code, the interface specification of the superclass must be reconstructed. When function symbol definitions are used during the verification of the client code, people in fact use some properties implied by these function symbol definitions. By investigating the verification, all of such properties can be retrieved. These properties, together with the function symbols and the method specifications, compose an interface specification. If a subclass implements this interface, it can substitute the superclass without re-verifying the client code.
To design a subclass which can substitutes the superclass in any client code. Suppose that the predicate symbol INV are defined as INV\textsubscript{sup} and INV\textsubscript{sub} in the superclass and the subclass respectively. For each method m, the pre-condition and post-condition of m are INV\textsubscript{sup} \land P_{m, sup} and INV\textsubscript{sup} \land Q_{m, sup} in the superclass; and INV\textsubscript{sub} \land P_{m, sub} and INV\textsubscript{sub} \land Q_{m, sub} in the subclass. We can conclude that the subclass can be used in any place the superclass is used if the following conditions holds.

1. For each method m of the superclass, it holds that
   \[(\text{INV}_{\text{sub}} \land P_{m, \text{sup}} \Rightarrow P_{m, \text{sub}}) \land (\text{INV}_{\text{sub}} \land Q_{m, \text{sub}} \Rightarrow Q_{m, \text{sup}})\]
2. The objects of both the superclass and the subclass do not share their private memories with other objects.
3. The definition of the predicate INV is not used during the verification of the client code.

Let B be a superclass, and D be a subclass of B. Because of the condition 2 the invariant of an object always holds when a method of the object is invoked. From the condition 1 and 3 at any place the precondition of $B::m$ holds, the precondition of $D::m$ also holds. So B can be substituted with D in any client code of B. This is in fact a variant of the Liskov Substitution Principle.

Now we show that these conditions are a special case of the principle presented in this paper to some extent: for an arbitrary piece of client code using the superclass, we can construct an interface implemented by both the superclass and the subclass.

- All the function symbols (including INV) defined in the superclass are declared in the interface. All the properties about these symbols used in the verification of the client code are expressed as constraint templates.
- For each method m, two function (predicate) symbols are declared: PreSym\textsubscript{m} and PostSym\textsubscript{m}. The specification template of m in the interface is
  \[
  \{\text{INV} \land \text{PreSym}_{m}\}\{\text{INV} \land \text{PostSym}_{m}\}
  \]

  There are two constraint templates about these two symbols.

  \[
  \begin{align*}
  \text{INV} \land P_{m, \text{sup}} & \Rightarrow \text{PreSym}_{m} \\
  \text{INV} \land \text{PostSym}_{m} & \Rightarrow Q_{m, \text{sup}}
  \end{align*}
  \]

  Because this interface contains all the properties about the symbols, the verification of the client code can be done based on this interface specification. It can be checked that both the superclass and the subclass implement this interface:

- The superclass defines PreSym\textsubscript{m} and PostSym\textsubscript{m} as $P_{m, sup}$ and $Q_{m, sup}$ respectively, while the subclass defines these two function symbols as $P_{m, sub}$ and $Q_{m, sub}$ respectively. From the conditions 1 the constrains about PreSym\textsubscript{m} and PostSym\textsubscript{m} are satisfied by both the superclass and subclass.
class SetByListWSize
{
var: Node * head;
funcs:
  attrib SetOf(int) theSet() ≜
    λx.x → D[NodeSet(head)];
  attrib bool INV() ≜
    isSList(head) ∧
    size = sizeOf(theSet());
  attrib SetOf(Ptr) pmem() ≜
    {&this→head} ∪
    (λx.(&x → D)[NodeSet(head)]) ∪
    (λx.(&x → link)[NodeSet(head)]);
method:
  SetByListWSize() pre ρ ∧ INV() post ρ ∧ INV() ∧
    (theSet() = ∅)
  {
    head = nil;
    size = 0;
  }
  bool isIn(int x) pre ρ ∧ INV() post ρ ∧ INV() ∧
    (ret = (x ∈ this→theSet()))
  {
    Node *cur;
    cur = head;
    while(cur ≠ nil ∧ cur→D ≠ x)
      cur := cur→link;
    return (cur = nil);
  }
  int getSizeOf() pre ρ ∧ INV() post ρ ∧ INV() ∧
    (ret = sizeOf(this→theSet()))
  {
    return size;
  }
  void Add(int x) pre ρ ∧ (M(ρ) ∩ pmem() = ∅) ∧
    INV() ∧ (x ∉ this→theSet()) post ρ ∧ INV() ∧
    (theSet() = theSet() ∪ {x})
  {
    Node *tmp;
    tmp := alloc(Node);
    tmp→D = x; tmp→link = head;
  }
};
}

class SetByListWSize inherit SetByList
{
var: Node * head; int size;
funcs:
  attrib SetOf(int) theSet() ≜
    λx.x → D[NodeSet(head)];
  attrib bool INV() ≜
    isSList(head) ∧
    size = sizeOf(theSet());
  attrib SetOf(Ptr) pmem() ≜
    {&this→head} ∪
    (λx.(&x → D)[NodeSet(head)]) ∪
    (λx.(&x → link)[NodeSet(head)]);
method:
  SetByListWSize() pre ρ ∧ INV() post ρ ∧ INV() ∧
    (theSet() = ∅)
  {
    head = nil;
    size = 0;
  }
  bool isIn(int x) pre ρ ∧ INV() post ρ ∧ INV() ∧
    (ret = (x ∈ this→theSet()))
  {
    Node *cur;
    cur = head;
    while(cur ≠ nil ∧ cur→D ≠ x)
      cur := cur→link;
    return (cur = nil);
  }
  int getSizeOf() pre ρ ∧ INV() post ρ ∧ INV() ∧
    (ret = sizeOf(this→theSet()))
  {
    return size;
  }
  void Add(int x) pre ρ ∧ (M(ρ) ∩ pmem() = ∅) ∧
    INV() ∧ (x ∉ this→theSet()) post ρ ∧ INV() ∧
    (theSet() = theSet() ∪ {x})
  {
    Node *tmp;
    tmp := alloc(Node);
    tmp→D = x; tmp→link = head;
    size = size + 1;
  }
};

Fig. 6. The class SetByList and its subclass SetByListWSize
– The superclass and the subclass give the same definitions to all other function symbols. So other constraints are also satisfied.

**Example 12.** Suppose that there is a class `SetBySList` as depicted in the left part of Fig. which records all the elements of a finite integer set in a singly linked list. The invariant `INV()` says that the member variable `head` refers to the head node of a singly-linked list. The `SetBySList` objects do not share their private memories with other objects.

To accelerate the queries on the size of the set, we can replace the class `SetBySList` by a subclass `SetBySListWSize`, depicted in the right part of Fig. This subclass has an extra member variables `size`. The invariant `INV()` says that the value of `size` is just the size of the set. This property is maintained by all the methods that modify the set. The specifications of the corresponding methods are literally identical, but the definitions of the symbols are different. The bodies of these methods are modified w.r.t. the new definitions. It can be checked that the relation between the subclass `SetBySListWSize` and the superclass `SetBySList` satisfies the conditions given in this subsubsection. So the subclass `SetBySListWSize` can be used in anyplace the superclass `SetBySList` is used.

If we found the extra cost of maintaining the value of `size` is higher than that saved by directly returning `size` in `getSize()` of `SetBySListWSize`, we can get a subclass `SetBySListWoSize` of `SetBySListWSize`. This subclass does not use the member variable `size` at all, and the methods are same as the ones in the class `SetBySList`. In this case, the post-conditions of the methods of `SetBySListWoSize` is weaker than those of `SetBySListWSize`. The principle in this paper shows that `SetBySListWoSize` is still a subtype of `SetBySListWSize`. □

9 Related Works and Conclusions

‘Programming to interfaces’ is an important programming paradigm in Object-Oriented programming community. This paradigm is mainly supported by the runtime-binding facility through class-inheritance or interface-implementation in OO programming languages. An invocation to a method of an interface (or superclass) may be dynamically bound to a method defined in an implementing class (or a subclass). The main challenge to specify ‘programming to interface’ code is to deal with the polymorphism caused by the runtime-binding facility.

Many research works have been proposed to deal with the polymorphism cased by method inheritance and overriding. Most of the works use the LSP (Liskov Substitution Principle) subtyping rule to avoid re-verification of the client code. Once a method has committed to a pre-condition/post-condition contract, any redefinition of this method through overriding must preserve to this commitment.

In [6], the Virtual Method approach is presented to handle method overriding and dynamic dispatch. Virtual methods represent the common properties of all
corresponding subtype methods. All the implementations of a virtual method must commit to these abstract properties defined in the superclass. However, these abstract properties may be too weak to verify the client code using the superclass, especially when the superclass is abstract. In [1], each method is associated with two specifications: a static specification, which is applied when the dynamic class of the receiver is known, and a dynamic one, which is used for method invocation with dynamic dispatch. A dynamic specification must be a specification supertype of its static counterpart, and be a specification supertype of the dynamic specification of each overriding method in its subclasses. A similar approach is presented in [4]. In [7], a set of axioms are used to specify that the overriding methods must commit to the specifications of the overridden ones. All the above approaches deal with dynamic dispatch based on the LSP sub-typing rule (or its variants). As we discussed before, such approaches are not suitable for the ‘programming to interfaces’ paradigm, because the interfaces (or abstract classes) declare no member variable. The interface methods can not be precisely specified without member variable.

In [2][3], a lazy form of behavioral sub-typing is presented. The behaviors of the overriding methods are not restricted by the specifications of the overridden methods. The overriding methods are only required to preserve the ‘part’ of the specifications that actually used to verify the client codes. This approach still requires the specifications of the overridden methods. So it is not powerful enough to deal with the ‘programming to interfaces’ paradigm.

Abstract predicate families are used in [5] to specify the methods of superclasses and subclasses. These predicates can also be re-defined differently by subclasses. However, they are mainly used to deal with the extend fields in subclasses. In our paper, abstract function symbols are used to capture the relationship between interfaces and classes.

In this paper, we present a flexible and precise approach to specify and verify code written in the ‘programming to interfaces’ paradigm. It addresses the following problems.

– **How to specify an interface.** A set of abstract function/predicate symbols, together with a set of constraints on these symbols, are declared in the interface. The methods of the interface can be precisely specified using these function/predicate symbols.

– **How a class implements an interface.** The class can gives its own definitions to the function/predicate symbols in the interface, as long as the constraints declared in the interface are satisfied. The class can give its own implementations to the methods declared in the interface, as long as the method specifications are satisfied w.r.t. the symbol definitions in the class.

– **How to verify the client code using interfaces.** The term `classOf(e)` is used to denote the runtime class of `e`. Given an expression `e`, `classOf(e)::f` refers to the function definition of `f` in the class `classOf(e)`. Logical deduction can be performed based on the constraint templates declared in the interfaces, without referring to the classes implementing these interfaces.
The class-interface-implementation relation proposed in this paper is flexible. A superclass is viewed as an interface plus an implementation to this interface. From this view, if a subclass is a subtype of the superclass according the LSP subtyping rule, we can add some function symbols to the interface of the superclass such that the subclass implements the interface according to the class-interface-implementation relation proposed in this paper. To some extent, the LSP subtyping rule is a special case of our class-interface relation.

While the interface method specifications are abstract, they are precise enough to deal with some popular real-world examples (See Appendix). The client code can be specified and verified without referring to the implementing classes. However, when more information about the dynamic classes of expressions are known, the client code specifications can be specialized to more precise ones without re-verification. In the ‘programming to interfaces’ paradigm, programmers can give different implementations to an interface to make the client code fulfill different functional features. The approach presented in this paper supports this advantage in that different specifications of a piece of client code can be inferred from the general specification of the client code and the different implementations of the interface.

We also give several real-world examples in the Appendixes to demonstrate the power of our approach.

References

1. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. In George C. Necula and Philip Wadler, editors, POPL, pages 87–99. ACM, 2008.
2. Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Lazy behavioral subtyping. J. Log. Algebr. Program., 79(7):578–607, 2010.
3. Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. Sci. Comput. Program., 76(10):915–941, 2011.
4. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst., 16(6):1811–1841, November 1994.
5. Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL ’08, pages 75–86, New York, NY, USA, 2008. ACM.
6. Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential java. In S. Doaitse Swierstra, editor, ESOP, volume 1576 of Lecture Notes in Computer Science, pages 162–176. Springer, 1999.
7. Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. Automatic verification of java programs with dynamic frames. Formal Asp. Comput., 22(3-4):423–457, 2010.
8. Jianhua Zhao and Xuandong Li. Scope logic: An extension to hoare logic for pointers and recursive data structures. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, ICTAC, volume 8049 of Lecture Notes in Computer Science, pages 409–426. Springer, 2013.
A The specification of the interface Comparator

In this section, we given the specification of the interface ComparatorOfPoints, which can be used to compare two points. The interface ComparatorOfPoints declared in Fig. 7 is similar to the interface java.util.Comparator. The interface java.util.Comparator is a template and has a type parameter T. As templates are not supported by the small language in this paper, the parameter T is fixed as Point, so we have ComparatorOfPoints. Two implementations of this interface are also given, together with a piece client code using this interface.

A.1 The interface specification

The specification of the interface ComparatorOfPoints is given in Fig. 4. It declares an object-function symbol LE. The first three constraint templates declared in the interface say that LE induces a total order. The fourth constraint template says that the memory scope of LE is a subset of the private memories of this and the two points being compared. The method Compare compares two points w.r.t. the order induced by LE.

```plaintext
interface ComparatorOfPoints
{
  funcs:
    bool LE(Point v1, Point v2);
  cons:
    ∀ v : Point. (v ≠ nil ⇒ LE(v, v));
    ∀ v1, v2 : Point. (v1 ≠ nil ∧ v2 ≠ nil ⇒ (LE(v1, v2) ∨ LE(v2, v1)));
    ∀ v1, v2, v3 : Point. (LE(v1, v2) ∧ LE(v2, v3) ⇒ LE(v1, v3));
    ∀ v1, v2 : Point. (v1 ≠ nil ∧ v2 ≠ nil ⇒ (2M(LE)(v1, v2) ⊆ (this → pmem() ∪ v1 → pmem() ∪ v2 → pmem())));
  methods:
    int Compare(Point *o1, Point *o2);
    Pre { ρ ∧ o1 ≠ nil ∧ o2 ≠ nil }
    Post { ρ ∧ (ret ≤ 0 ⇒ LE(o1, o2)) ∧ (ret ≥ 0 ⇒ LE(o2, o1)) }
}
```

Fig. 7. The specification of the interface ComparatorOfPoints

A.2 Two implementations of ComparatorOfPoints

In this subsection, we will give two different implementations of the interface ComparatorOfPoints. The first implementation disComp compares two Point objects by their distances from a given point, and the second implementation XYComp first compares two Point objects by their x-coordinates, and then compares their y-coordinates if two points have the same x-coordinate. These two implementations are depicted in Fig. 8. We can check that these two classes implement the interface ComparatorOfPoints.
class ArraySort
{
var:
Point arr[10];

functions:
SetOf(Point psem) ≜ \lambda x.(\&this->arr[x])[0..9];
Point get(int i) ≜ arr[i];

methods:
void Sort(ComparatorOfPoints cmp)
Pre \( \rho \land (\exists x \in 0..9.\neg get(x) \neq nil)) \land MemoryLayout
Post \( \rho \land \forall i \in [0..9].(\neg cmp->psem() \land this->psem() = \emptyset) \land \lambda x.(\&arr[x] \to get(i)) \land (\&this->psem() = \emptyset)

\begin{align*}
\text{The invariant of the outer loop is} & \\
\rho \land \text{MemoryLayout} \land \\
\forall x \in (1..9).&(\neg cmp->LE(arr[x], arr[x+1]))
\end{align*}

\begin{align*}
\text{The invariant of the inner loop is} & \\
\rho \land \text{MemoryLayout} \land \\
\forall x \in (0..j-1).&(\neg cmp->LE(arr[x], arr[x+1]))
\end{align*}

Fig. 8. Tow implementations of the interface ComparatorOfPoints

class XYComp impl ComparatorOfPoints
{
functions:
bool LE(Point o1, Point o2) ≜
(\&o1->FldX() < \&o2->FldX()) \lor \\
(\&o1->FldX() = \&o2->FldX()) \land \\
\&o1->FldY() \leq \&o2->FldY()

methods:
XYComp( )
Pre \( \rho \land (\exists x \in 0..9.\neg this->BLOCK()) = \emptyset) \\
Post \( \rho \land \neg \exists x \in 0..9.\neg this->BLOCK())

int Compare(Point *o1, Point *o2)
{
int x1, int y1, 
int x2, int y2;

x1 = \&o1->getX(); y1 = \&o1->getY();

x2 = \&o2->getX(); y2 = \&o2->getY();

return \((-x1 + x2)^2 + (y1 - y2)^2 \lesssim (x2 - x1)^2 - (y2 - y1)^2\)

\}

\}

Fig. 9. The sorting algorithm using Comparator, and its specification

MemoryLayout is the abbreviation for
(\exists x \in 0..9.\neg get(x) \neq nil)) \land MemoryLayout
\forall x \in (1..9).&(\neg cmp->psem() \land this->psem() = \emptyset) \land \\
\lambda x.(\&arr[x] \to get(i)) \land (\&this->psem() = \emptyset)

\begin{align*}
\text{The invariant of the outer loop is} & \\
\rho \land \text{MemoryLayout} \land \\
\forall x \in (1..9).&(\neg cmp->LE(arr[x], arr[x+1]))
\end{align*}

\begin{align*}
\text{The invariant of the inner loop is} & \\
\rho \land \text{MemoryLayout} \land \\
\forall x \in (0..j-1).&(\neg cmp->LE(arr[x], arr[x+1]))
\end{align*}
A.3 The client code using ComparatorOfPoint

The class ArraySort depicted in Fig. 9 has a method Sort which compares the Point objects in the list using the interface ComparatorOfPoints.

The invariants of the two while-statements are depicted on the right part of Fig. 9. Using these invariants, the function can be verified based on the properties about LE presented in Fig. 7. We can sort the objects in the array arr by invoking the method Sort using different comparator as parameter. If the real parameter of Sort is the object created by the following statement

\[
\text{cmp} = \text{new disComp}(5, 4);
\]

the Point objects in arr are sorted according to the distances from the point (5, 4). If the real parameter is created by

\[
\text{cmp} = \text{new XYComp}();
\]

the Point objects are sorted according to the X-coordinates first, and then the Y-coordinates for objects with the same X-coordinates.
B The Set interface and its Iterator interface

The Iterator pattern decouples the codes traversing elements in a container from the implementation of the container. This section shows how to specify an Iterator over a set container.

![Fig. 10. The interfaces SetOfInt and Iterator](image)

B.1 The specification of the interfaces

The interface specifications of Iterator and SetOfInt is given in Fig. 10. The interface SetOfInt has an attribute function symbol `theSet`, which denotes the set represented by the object. The methods for set operations, e.g. add, delete, are omitted. The method `createIterator` creates an iterator of this object. The interface Iterator declares 3 function symbols.

- `theContainer()` denotes the set to which this iterator is attached;
- `passedSet()` denotes the set of integers in `theContainer()` that have been traversed by the method `Next()`;
- `SOUND()` specify that `theContainer()` refers to a valid object of the set.
The method `hasNext()` test whether all the elements are traversed. The method `Next()` returns an element that has not been traversed.

### B.2 The classes implementing the interfaces

Each class implementing the interface `SetOfInt` should have a method to create corresponding iterators. An iterator can be used to traverse all the elements in the set that creates this iterator. The class `SingleListSetOfInt` given in Fig. 11 implements the interface `SetOfInt`. Its corresponding iterator class `IteratorOfSLS`, which is an implementation of the interface `Iterator`, is also given in Fig. 11.

```plaintext
class SetOfIntSL impl SetOfInt
{
  var:
    Node *head;
  funcs:
    attribute SetOf(int) theSet() ≜ λx.x→data[NodeSet(head→link)];
    bool INV() ≜ IsSList(head) ∧ head ≠ nil;
    SetOf(Ptr) pmem() ≜ λx.x→link[NodeSet(head)];
    Node * getHead() ≜ head;
  method:
    ... ... ...
    int createIterator()
    {  Iterator it = new IteratorOfSLS(this, head);  return it;  }
}

class IteratorOfSLS impl Iterator
{
  var:
    SetOfIntSL thecontainer;
    Node *cur;
  funcs:
    bool SOUND() ≜ thecontainer ≠ true ∧ thecontainer→INV();
    SetOfInt theContainer() ≜ thecontainer;
    SetOf(int) passedSet() ≜ theContainer()→theSet() – λx.x→data[NodeSet(cur→link)];
  method:
    IteratorOfSLS(SetOfIntSL *container, Node *head)
    pre  ρ ∧ (∃N(ρ) ∧ pmem() = ∅) ∧ container→INV() ∧ head = container→getHead();
    post  ρ ∧ SOUND() ∧ theContainer() = container ∧ passedSet() = ∅
    {  thecontainer = container;  cur = head;  }
    bool hasNext() { return (cur→link ≠ nil);}  // cur = cur→link;  return cur→data; }
}
```

**Fig. 11.** The classes implementing `SetOfInt` and `Iterator`
B.3 The client code using the interfaces

A piece of client code using the interface `Iterator` is given in Fig. 12. The invariant of the while statement is

\[\begin{align*}
\text{set} &= \text{it} \rightarrow \text{theContainer()} \land \text{it} \rightarrow \text{SOUND()} \\
\text{bNotFinished} &= (\text{it} \rightarrow \text{passedNet()} \subseteq \text{set} \rightarrow \text{theSet()} ) \land \\
\text{bHasNegative} &= (\forall x \in \text{it} \rightarrow \text{passedSet()}.(x \geq 0))
\end{align*}\]

It can be proved that the post-condition of this code segment is

\[\text{bHasNegative} == (\forall x \in \text{set} \rightarrow \text{theSet()}.(x \geq 0))\]

```cpp
SetOfInt set; Iterator it; bool bHasNegative; bool bNotFinished;

set = new SetOfIntSL();
set->add(1);
... ...
it = set->createIterator();
bHasNegative = false;
bFinished = it->hasNext();
while (bNotFinished)
{
    k = it->Next();
    if (k < 0)
        bHasNegative = true;
    else
        skip;
    bNotFinished = it->hasNext();
}
```

Fig. 12. The context using the `Iterator` interface and the `SetOfInt` interface
C The observer pattern

In this section, we will present the specifications and implementations of a simplified version of the class Observable and the interface Observer in the Java package java.util. Roughly speaking, Observable and Observer work as follow. An Observable object is attached with a set of Observer objects. If the Observable object is modified, the method notifyObservers is invoked to notify all the Observer objects. For each Observer object, the method updateData() is invoked such that the Observer object is accordingly modified to maintain a consistency relation with the Observable object. In this design pattern, the implementations of Observer are dependent on the implementation of Observable. That is, an implementation of Observer can only observe a specific implementation of Observable.

In the package java.util, Observable is a class and some basic methods are already implemented. People should implement a subclass of Observable to fulfill some expected functional features. Because inheritance is not supported in the small language, Observable is declared as an interface and the class implementing Observable should implement the basic methods by itself. This section can also be viewed as an example of dealing with subclassing and inheritance: we can specify the interface of a superclass, and make the subclasses implement the interface.

C.1 The specifications of the interfaces Observable and Observer

The interfaces Observable and Observer are specified in Fig. 13. The interface Observable declares two function symbols.

− the attribute function symbol theObservers yields the set of observers.
− the object-function symbol bindObservers() specify that the private memories of the observers are disjoint with each other.

Two methods are declared in the interface: addObserver and notifyObservers. The first one adds a new observer into the observer set; the second one is to be invoked when the Observable object is modified, and all the observers must update their internal data to keep consistent with the Observable object.

The interface Observer declares two function symbols. The function symbol bCanObserve tests whether this observer can observe the subject; the function symbol bConsistentWith specifies that the observer keeps consistent with the subject being observed. The method Update() modifies the internal data of this observer to keep consistent with the real parameter.

C.2 The implementations of Observable and Observer

In this subsection, we give a class TextModel implementing Observable, and two classes, LengthObserver and LineNumObserver, implementing Observer. A TextModel object holds a string. The methodSetText sets the text string of
interface Observable
{
  funs:
    attribute SetOf(Observer) theObservers();
    bool blndObservers();
  cons:
    blndObservers() ⇒ ∀x, y ∈ theObservers().(x ≠ y ⇒ x→pmem() ∩ y→pmem() = ∅);
  methods:
    ... ...
      void addObserver(Observer ob);
      pre {ρ ∧ (M(ρ) ∩ M(theObservers()) = ∅) ∧ ob→bCanObserve(this) ∧ blndObservers() ∧ ∀x ∈ theObservers().(o→pmem() ∩ x→pmem() = ∅)}
      post {ρ ∧ (theObservers() = theObservers() ∪ {ob}) ∧ blndObservers()}
    void notifyObservers( );
      pre {ρ ∧ ∀x ∈ theObservers().(M(ρ) ∩ x→pmem() = ∅) ∧ blndObservers()}
      post {ρ ∧ ∀x ∈ theObservers().(x→bConsistantWith(this)) ∧ blndObservers()}
}

interface Observer
{
  funs:
    bool bCanObserve(Observable *sub);
    bool bConsistantWith(Observable *sub);
  cons:
    ∀x : Observable(bCanObserve(x) ⇒ M(bConsistantWith(x)) ⊆ pmem() ∪ x→pmem())
  methods:
    void updateData(Observable *sub);
      pre {ρ ∧ (M(ρ) ∩ pmem() = ∅) ∧ (sub ≠ nil) ∧ bCanObserve(sub)}
      post {ρ ∧ bConsistantWith(sub)}
}

Fig. 13. The interfaces Observable and Observer
class TextModel impl Observable
{
  var:
  String str;
  SetOfObserver obSet;
funcs:
  attribute SetOf(Observer) theObservers() ꩄ obSet→theSet();
  bool bindObservers() ꩄ ∀x, y ∈ theObservers(),(x ≠ y ⇒ x→pmem() ⊆ y→pmem() = Ø);
  String getTextString() ꩄ str;
methods:
  TextMode() pre { ρ } post { ρ ∧ (theObservers() = ∅) ∧ getTextString() = " " } {
    str = " "; obSet = new SetOfObserverSL(); }
  void SetText(String s) pre { ρ ∧ (M(ρ) ∩ pmem() = ∅) } post { ρ ∧ (theObservers() = ←−−−−−−−−−−theObservers()) ∧ getTextString() = s } {
    str = s; }
  . . . . . .
  void addObserver(Observer ob); {
    obSet→AddEle(ob); }
  void notifyObservers( ); {
    Iterator it; Observer o; bool tmp;
    it = obSet.createIterator();
    tmp = it.hasNext();
    while (tmp)
    { o = it.Next(); o→updateData(this); tmp = it.hasNext(); }
  }
}

class LengthObserver impl Observer
{
  var:
  int len;
funcs:
  int length() ꩄ len;
  bool bCanObserve(Observable sub) ꩄ (classOf(sub) = TextModel);
  bool bConsistantWith(Observable sub) ꩄ (length() = strlength(sub→getTextString()));
methods:
  LengthObserver() pre { ρ } post { ρ ∧ len = 0; } {
    len = 0; }
  void updateData(Observable sub) {
    String s; TextMode tm;
    tmp = (TextModel)sub; s = tm→getTextString(); len = s→length();
  }
}

class LineNumObserver impl Observer
{
  var:
  int lineNum;
funcs:
  int line() ꩄ lineNum;
  bool bCanObserve(Observable sub) ꩄ (classOf(sub) = TextModel);
  bool bConsistantWith(Observable sub) ꩄ (line() = NumOfSubString(sub→getTextString(), "\n"));
methods:
  LineNumObserver() pre { ρ } post { ρ ∧ lineNum = 0; } {
    lineNum = 0; }
  void updateData(Observable sub) {
    String s; TextMode tm;
    tmp = (TextModel)sub; s = tm→getTextString(); lineNum = s→NumOfSubStr("\n");
  }
}

Fig. 14. The implementations of Observable and Observer
a TextModel object. An object of LengthObserver (or LineNumObserver) can observe a TextModel object. It records the number of lines (or the length of the text string) of the TextModel object being observed.

A container class SetOfObserverSL and its iterator IteratorOfSObLS is used to record and manipulate the set of observers. Their specifications and implementations are similar to the ones in Section [4]. The String class represents character strings. The method length() returns the length of the string, the method NumOfSubStr returns the number of occurrences of the real parameter in the string.

C.3 The client code using the Observable and Observer

The piece of code depicted in Fig. [15] using these two interfaces and their implementations. It first creates a TextModel object, a LengthObserver object, and a LineNumObserver object. The later two objects are set to observe the TextModel object. After some manipulations are performed on the TextModel, the method NotifyObservers is invoked. After invocation, these two observers are consistent with the TextModel object, i.e. the following property holds.

lenOb → ConsistentWith(tm) ∧ lnOb → ConsistentWith(tm)

Because the runtime classes of lenOb and lnOb are respectively LengthObserver and LineNumObserver, the above property is equivalent to

(lnOb → lNo() = NumOfSubString(sub → getTextString(), "\n")(lenOb → length() = strlength(sub → getTextString()))

Fig. 15. A piece of code using the interface Observer and Observable