Specifying and Verifying Object-Oriented Programs: An Overview of the Problems and a Solution

Gary T. Leavens
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Systems Architecture Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/94

This Article is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
Specifying and Verifying Object-oriented Programs:
An Overview of the Polenada Solution

TR91-06
Gary T. Leavens
February, 1991
Specifying and Verifying
Object-Oriented Programs:
an Overview of the Problems and
a Solution

Gary T. Leavens

TR #91-06
February, 1991

This paper will appear in IEEE Software. © 1991 IEEE

Keywords: verification, specification, subtype, message passing, polymorphism, type checking, modularity.


Department of Computer Science
Iowa State University
Ames, Iowa 50011-1040, USA
Specifying and Verifying Object-Oriented Programs:
an Overview of the Problems and a Solution

Gary T. Leavens*
Department of Computer Science
Iowa State University, Ames, Iowa 50011 USA
leavens@cs.iastate.edu (515)294-1580
February 10, 1991

Abstract
This paper presents a careful analysis of the problem of reasoning about object-oriented programs. A solution to this problem allows new types to be added to a program without respecifying or reverifying unchanged modules — if the new types are subtypes of existing types. The key idea is that subtype relationships must satisfy certain semantic constraints based on the types’ specified behavior. Thus subtyping is not the same as inheritance of implementations (subclasing).

Subtyping aids specification and verification of object-oriented programs by allowing supertypes to stand for their subtypes. This reduces the problem of reasoning about both supertypes and their subtypes to the problems of reasoning about just the supertypes and proving that the subtype relationships satisfy the required constraints.

1 Introduction

Abstraction allows one to ignore unimportant details in reasoning. Not only does abstraction make arguments more succinct, but it also allows arguments to depend on weaker assumptions. For example, one reasons about an abstract data type according to its specification, ignoring details of how its objects are represented [1]. A less well-known use of abstraction, but one that is important in object-oriented programming methods, is the use of supertypes as abstractions of their subtypes. For example, windows may stand for bordered windows or menus. Supertypes can stand for their subtypes during specification and verification, if they are used in a disciplined fashion. Having supertypes stand for their subtypes is called supertype abstraction.

Subtypes should not be confused with subclasses [2]. A class is a program module that implements an abstract data type. A subclass inherits data representation and operations from its superclasses, but a subclass may also change inherited aspects. A type is an

*This work was supported in part by the National Science Foundation under Grants DCR-8510014 and CCR-8716384, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, a GenRad/AEA Faculty Development Fellowship, and by the ISU Achievement Foundation.
abstraction of several classes, characterized by a behavioral specification. A subtype specializes the specification of one or more types. Thus a subclass relationship is a relationship between implementation modules, while a subtype relationship is a relationship between specifications. In general, a subclass does not implement a subtype and a subtype need not be implemented by a subclass.

The problems and benefits of supertype abstraction are best illustrated in the context of a common program enhancement: adding a new type of data to an existing program. For example, one might add borders to the windows in a window system, menus as a special case of windows, pop-up menus as a special case of menus, priorities to the queues of an operating system, and so on. When does adding a new data type not cause problems for existing modules? How can one design new types so that they will not cause problems in existing software? The folklore is that if the new types are “subtypes” of one or more existing types, then the program will work without problems [2]. For example, if bordered windows are a subtype of windows, then existing code should not have problems when bordered windows are used in place of windows. The goal of the following investigation is to present these problems analytically, and to informally explain the insights into software design and verification that are the fruits of a formal definition of what it means to be a “subtype.”

From an analytical perspective, the problem is how to formally specify and verify object-oriented programs in a modular fashion. A specification and verification method is modular if when new types of objects are added to a program, the specifications and verifications of existing types, functions, and their implementations do not have to be redone. Even if you are not concerned about formal specification and verification of programs, this problem is important, because knowing when specifications and verifications have to be changed allows you to know when to rethink existing modules. In situations where the formal verification of an existing implementation is sound without change, you can be confident that existing modules will work as well as before, even if they have not been formally verified. In this sense formal methods can help guide informal reasoning about programs and can give guidance in subtle situations.

2 An Example

To illustrate the problems supertype abstraction causes for reasoning, consider a system to keep track of which keys unlock certain doors and who has what keys. One can represent key numbers by objects of type \texttt{Int} (integers), and the set of keys possessed by a person as an object of type \texttt{IntSet}. The operations of these types can be used to perform such
tasks as recording issued and returned keys, and finding whether two people have any keys in common.

The type IntSet is specified in Figure 1. The type specification describes the behavior of the following operations:

- **null**, which creates an empty IntSet,
- **ins**, which returns an IntSet containing its integer argument inserted into the set of elements of its IntSet argument,
- **elem**, which tests whether an integer is in an IntSet,
- **choose**, which returns an arbitrary element of a nonempty IntSet,
- **size**, which returns the size of an IntSet, and
- **remove**, which returns an IntSet containing all the elements of its IntSet argument except for its integer argument.

None of the operations of IntSet changes the state of an existing IntSet. Since the objects have no time-varying state they are immutable. A type whose objects are all immutable is itself said to be immutable. (The formal results described below are limited to immutable types, although the general ideas also apply to programs that use mutable types.)

The formal specification of IntSet is given in a Larch-style interface specification language [3]. The trait IntSetTrait (see Figure 2) specifies the abstract values of IntSet objects (mathematical sets), and gives meaning to the trait function symbols $\cup$, $\in$, and so on that are used in the pre- and post-conditions of the program operations null, ins, and so on. Trait functions cannot be called from programs, and program operations cannot be used in pre- and post-conditions. (Program operations will usually be called “operations”, except when it is necessary to distinguish them from trait functions.)

After the keys system has been running for some time, one might want to extend it so that it can issue a set of keys with consecutive numbers. Since such a set can be represented in less storage than a general set, it may be wise to add a new type to the design. This is the type Interval (see Figure 3). The abstract values of Intervals are specified in the trait IntervalTrait (see Figure 4). The operations of the type Interval are the same as those for IntSet, except that instead of null there is an operation create that takes two integer arguments and returns an Interval object representing all the integers between the arguments (inclusive). The arguments of create must be ordered. The choose operation of Interval always returns the least element of the Interval. The ins and remove operations
IntSet immutable type

class ops [null]
instance ops [ins, elem, choose, size, remove]
    based on sort C from IntSetTrait

    op null(c: IntSetClass) returns(s: IntSet)
    ensures s == {}

    op ins(s: IntSet, i: Int) returns(r: IntSet)
    ensures r == (s ∪ {i})
    op elem(s: IntSet, i: Int) returns(b: Bool)
    ensures b = (i ∈ s)
    op choose(s: IntSet) returns(i: Int)
    requires ¬(isEmpty(s))
    ensures i ∈ s
    op size(s: IntSet) returns(i: Int)
    ensures i = toInt(size(s))
    op remove(s: IntSet, i: Int) returns(r: IntSet)
    ensures r == delete(s, i)

Figure 1: The type specification IntSet. Each operation of this type is specified after the keyword op. Each operation has a post-condition, which follows ensures. An operation may also have a pre-condition, which follows requires. The pre-condition defaults to “true”. When an operation’s pre-condition is not satisfied, it may either return any object of the appropriate type or not terminate.
\textbf{IntSetTrait: trait}

\begin{verbatim}
imports SetBasics with [Int for E],
    SetIntersection with [Int for E],
    isEmpty with [Int for E, {} for new],
    Singleton with [Int for E, {} for new, (#) for singleton],
    Join with [Int for E, {} for new, \& for join],
    CardToInt

introduces \#==\#: C,C \rightarrow \text{Bool}

asserts for all [s_1, s_2: C]
    (s_1 \#== s_2) = (s_1 = s_2)
\end{verbatim}

\textbf{CardToInt: trait}

\begin{verbatim}
imports Cardinal, Integer

introduces toInt: Card \rightarrow \text{Int}

asserts for all [c: Card]
    toInt(0) = 0
    toInt(succ(c)) = (1 + toInt(c))
\end{verbatim}

Figure 2: The traits \texttt{IntSetTrait} and \texttt{CardToInt}. The \texttt{imports} section brings in the text of the named traits (from [4]). The renamings following \texttt{with} alter the text of the imported traits; for example substituting \texttt{Int} for \texttt{E}. Trait functions and their signatures are declared after \texttt{introduces}. The trait function \texttt{\#==} is an infix operator, because its declaration uses sharp signs (\#) to show argument positions. Equations in the \texttt{asserts} section are universally quantified over all abstract values of the given types.
Interval immutable type
subtype of IntSet by \([l, u]\) simulates toSet\((l, u)\)
class ops [create]
instance ops [ins, elem, choose, size, remove]
based on sort C from IntervalTrait

\begin{align*}
\text{op } & \text{create(c:IntervalClass, lb,ub:Int) returns(i:Interval)} \\
& \text{requires } lb \leq ub \\
& \text{ensures } i == [lb, ub] \\
\text{op } & \text{ins(s:Interval, i:Int) returns(r:IntSet)} \\
& \text{ensures } r == (s \cup \{i\}) \\
\text{op } & \text{elem(s:Interval, i:Int) returns(b:Boolean)} \\
& \text{ensures } b = (i \in s) \\
\text{op } & \text{choose(s:Interval) returns(i:Int)} \\
& \text{ensures } i = \text{leastElement}(s) \\
\text{op } & \text{size(s:Interval) returns(i:Int)} \\
& \text{ensures } i = \text{toInt(size(s))} \\
\text{op } & \text{remove(s:Interval, i:Int) returns(r:IntSet)} \\
& \text{ensures } r == \text{delete}(s,i)
\end{align*}

Figure 3: The type specification Interval. This type is specified as a subtype of IntSet. The subtype relationship is justified in the by clause, which says what IntSet abstract value an interval with abstract value \([l, u]\) simulates. The trait function “toSet” is defined in Figure 4, along with the abstract values and trait functions used in the pre- and post-conditions of the operations.

of must be allowed to return IntSet objects; consider inserting 99 in the interval \([12, 15]\). But because Interval is specified as a subtype of IntSet, ins and remove are also allowed to return Interval objects; for example, when removing 15 from \([12, 15]\), the interval \([12, 14]\) can be returned. This capability reflects consistent use of supertype abstraction in type specifications.

3 Subtype Polymorphism

Adding the type Interval to the keys system brings up the following problems. Does one have to update the code to work with Interval objects? How does one ensure the correctness of the updated code?

To eliminate the first of these problems, object-oriented programming languages provide objects and a message-passing mechanism. (Message passing is sometimes also called dynamic binding or late binding.) Conceptually, each object contains, in addition to its data,
**IntervalTrait**: trait

- **Imports**: IntSetTrait with [IntSet for C]
- **Introduces**: [#, #]: Int, Int → C
  - Insert, delete: C, Int → IntSet
  - Size: C → Card
  - #∈#: C, Int → Bool
  - isEmpty: C → Bool
  - union, ∩: C, C → IntSet
  - union, ∩: IntSet, C → IntSet
  - #≡#: C, C → Bool
  - #≡#: IntSet, C → Bool
  - toSet: C → IntSet

- **Least Element, Greatest Element**: C → Int

**Asserts for all** [c, c1: C, s: IntSet, x, y, i: Int]

\[
[x, y] = \begin{cases} [x, y] & \text{if } x \leq y \\ [x, x] & \text{else} \end{cases}
\]

- Insert: \([x, y], i = \text{insert}(\text{toSet}([x, y]), i)\)
- Delete: \([x, y], i = \text{delete}(\text{toSet}([x, y]), i)\)
- Size: \([x, y] = \text{size}(\text{toSet}([x, y]))\)
- \((i \in [x, y]) = (i \in \text{toSet}([x, y]))\)
- isEmpty([x, y]) = false
- leastElement([x, y]) = x
- \((s == c) = (s == \text{toSet}(c))\)
- \((c == s) = (s == \text{toSet}(c))\)
- \((c == c1) = (\text{toSet}(c) == \text{toSet}(c1))\)
- \((s \cap c) = (s \cap \text{toSet}(c))\)
- \((c \cap s) = (s \cap \text{toSet}(c))\)
- \((c \cap c1) = (\text{toSet}(c) \cap \text{toSet}(c1))\)
- \((s \cup c) = (s \cup \text{toSet}(c))\)
- \((c \cup s) = (s \cup \text{toSet}(c))\)
- \((c \cup c1) = (\text{toSet}(c) \cup \text{toSet}(c1))\)
- greatestElement([x, y]) = \begin{cases} [x, y] & \text{if } x \leq y \\ \{x\} & \text{else} \end{cases}
- toSet([x, y]) = \begin{cases} [x, y] & \text{if } y \leq x \\ \{x\} & \text{else} \end{cases}

**Figure 4**: The trait **IntervalTrait**. This trait adds to the definitions in **IntSetTrait**, whose text is included after changing occurrences of the type name C to **IntSet**, because in this trait the name “C” refers to intervals. This trait defines all the trait functions with the same names as those in **IntSetTrait** that act on **IntSet** arguments. For example, because “size” is defined on **IntSet** arguments, it is also defined here. For trait functions that take two arguments of type **IntSet**, three versions are defined here, so that each such trait function is defined on all combinations of **IntSet** and **Interval** arguments.
a mapping from pairs of operation names and types to code, called a method dictionary [5]. (For space efficiency the code for an operation is shared among all objects of the same type in most implementations of object-oriented languages.) Since the method dictionary is accessible from the objects, code that invokes an object’s operations does not have to depend on the types of objects. For example, one does not write `IntSet.ins(s, e)` to insert an integer `e` into a set `s`, as one would in Ada; instead, one writes `s.ins(e)` (in Simula 67 or C++) to insert `e` into `s`, which invokes the operation `ins` from the method dictionary of the object `s`. Thus message passing means fetching an object’s operation from its method dictionary and invoking it. Metaphorically `s.ins(e)` means “send the message ins with argument `e` to `s`.”

The advantage of using message passing is that `s.ins(e)` can invoke the `ins` operation of the types `IntSet`, `Interval`, and even types that have not yet been imagined. Thus code that works for `IntSet` objects does not have to be updated to work with `Interval` objects.

Code written using message passing is *polymorphic*, because it produces roughly the same effect on arguments of different types. For example, a function `inBoth` (see Figure 5) can find a key number that is common to two `IntSet` objects or two `Interval` objects (or an `IntSet` and an `Interval`) using the same sequence of message sends. However, the effect will be roughly the same only if the effect of these message sends on `Interval` objects is similar to their effect on `IntSet` objects; that is, a similar effect will be achieved only if `Interval` is a subtype of `IntSet`. I call this kind of polymorphism *subtype polymorphism*.

4 The Specification and Verification Update Problem

How should one reason about the behavior of a program to which new types of objects have been added? For example, suppose that, before adding the type `Interval` to the keys system, one has verified that the implementation of `inBoth` in Figure 5 is correct (when it is passed arguments of type `IntSet`). Does one have to go back and reverify the implementation of `inBoth` when it becomes possible to pass it arguments of type `Interval`? Since one does not have to update the code (because of message passing), it would be tiresome if one had to update the verification.

Furthermore, what does the specification of `inBoth` mean when the type `Interval` is added to the program? Consider the specification of Figure 6. Such a specification might be produced before the type `Interval` was contemplated. In Figure 6, the pre-condition and post-condition are expressed using trait functions, for example `∈`, `∩`, and “isEmpty”, from `IntSetTrait`. What does “`i ∈ s1`” mean if “`s1`” is an `Interval`? It would be tiresome if one had to update the specification of `inBoth` when new types were added to a program.
fun doSomething(s1,s2: IntSet): Int = 
inBoth(s1, s2.ins(6));
fun inBoth (s1,s2: IntSet): Int = 
testFor(s1.choose(), s1, s2);
fun testFor (i: Int, s1,s2: IntSet): Int = 
if s2.elem(i)
  then i
  else testFor((s1.remove(i)).choose(), s1.remove(i), s2)
fi;
program (b: Bool): Int = 
if b
  then doSomething((IntSet.null()).ins(3), Interval.create(2,5))
  else doSomething(Interval.create(1,4), Interval.create(2,5))
fi

Figure 5: Example of message passing. The main program consists of an if expression that calls the function doSomething with different arguments. The expression s1.choose() in the fourth line, i.e., in the body of inBoth, invokes an operation of IntSet or Interval, depending on the type of s1.

fun inBoth(s1,s2: IntSet) returns(i: Int) 
  requires ¬(isEmpty(s1 ∩ s2))
  ensures (i ∈ s1) & (i ∈ s2)

Figure 6: Specification of the function inBoth. The pre-condition follows requires. The post-condition follows ensures. The trait functions used in the pre- and post-conditions (e.g., “isEmpty”) are defined in the trait IntSetTrait, because both arguments have declared type IntSet. The identifiers “s1”, “s2” and “i” used in the pre- and post-conditions refer to the formal arguments and the result.

Respecification would also force re-verification.

The other side of the above specification problem is that to use Interval objects in a program, some part of the program must create new objects and pass them to existing functions such as doSomething. For example, to reason about the “main” program of Figure 5, one needs to show that the Interval objects it creates satisfy the pre-condition of doSomething (which for the purposes of this example can be assumed to be identical to the pre-condition of inBoth). The problem is that the post-condition of Interval’s create operation describes its result in the language of IntervalSetTrait, while the pre-condition of doSomething uses the language of IntSetTrait. To prove the needed implication, one must translate between these languages in a way that prevents misunderstanding. Once this
translation is accomplished, one can reason at a more abstract level, using the language of
the supertype and its specification.

5 Overview of a Solution

There are two main ideas for solving the specification and verification update problem. The
first is the notion of subtype relationships. A supertype must be able to stand for all its
subtypes during specification and verification. This implies strong constraints on the design
of subtypes of a given type. At the very least, each object of a subtype must behave like
some objects of the supertype, otherwise a program might behave in surprising ways when
it operates on some object of the subtype. Details on these constraints are discussed below.
If a type has multiple supertypes, then the constraints must hold between the type and
each of its supertypes.

The second idea is to use type checking to enforce a disciplined use of subtype polymor-
phism. The programmer, perhaps aided by the language's type system, statically assigns
each expression a type called its nominal type, with the property that the nominal type
is a supertype of the types of objects that the expression may denote at run-time. For
example, if \( s \) is declared to have nominal type \texttt{IntSet}, then \( s \) can denote an \texttt{Interval}
or an \texttt{IntSet}, but not an integer. The nominal type declared for an identifier is a supertype
of all the types of objects the identifier can denote at run-time; thus nominal types are
upper bounds instead of exact type information. Nominal types may be introduced solely
for program verification, or they may coincide with the types of the programming language.
The programming language's type system can be used if it can ensure that the nominal
type of each expression is an upper bound on the types of objects that the expression can
denote.

The use of types as upper bounds is an essential difference from standard program
verification techniques. Conventional verification techniques assume that at run-time each
expression of type \( T \) denotes an object created by a module that is an implementation of \( T \)'s
specification. This connection allows one to use the specification of type \( T \) to reason about
expressions of type \( T \).

However, to exploit subtype polymorphism, one must allow a given expression to denote
objects of several different types — that is, objects created by implementations of several
different type specifications. Otherwise much of the subtype polymorphism latent in a
program with message passing would remain unusable. So in a typed language with subtype
polymorphism, an expression of type \( T \) must be able to denote objects whose types are
subtypes of \( T \). Thus in a language with subtype polymorphism it is impossible, in general,
to statically determine the *exact* type of an object a given expression will denote at run-time.

The major benefit of using nominal types as upper bounds is that polymorphism can be limited to subtypes of a given type; for example, `inBoth` may only take arguments that are subtypes of the formal arguments’ nominal type (`IntSet`). Limiting arguments to subtypes and attaching semantic constraints to subtype relationships are crucial for the modular specification and verification of functions.

A method for reasoning about object-oriented programs that uses the ideas of subtype and nominal type was pioneered in the author’s dissertation and further developed in [6] [7]. The reasoning technique can be summarized as follows.

- One specifies the data types to be used in the program along with their subtype relationships.
- Procedures are specified by describing their effects on arguments whose types are the same as the types of the corresponding formal arguments; however, arguments whose types are subtypes of the corresponding formal argument types are permitted.
- Subtype relationships are verified to ensure that they satisfy the semantic constraints described below.
- Each expression in the program is statically given a nominal type. An expression of nominal type $T$ may only denote objects of a subtype of $T$.
- Verification that a program meets its specification is then nearly the same as conventional verification, despite the use of message passing. That is, one reasons about expressions as if they denoted objects of their nominal types. The exception occurs when one explicitly exploits subtyping, and for this case there is a simple verification rule.

When one adds a new type of data to a program, all that needs to be done is to specify that type and its subtype relationships, verify that the new type satisfies the semantic constraints for being a subtype, and verify any new or changed pieces of code. Unchanged functions and other types do not need to be respecified and reverified.

### 5.1 Subtyping

The key to the soundness of the method is a set of syntactic and semantic constraints on subtype relationships. These formalize the intuition that each object of a subtype must behave like some object of each of its supertypes. To discuss these constraints, it is first necessary to consider abstract data type specifications in more detail.
5.1.1 Abstract Type Specifications and Their Semantics

The specifications given above describe an abstract type in terms of a set of abstract values, trait functions, and pre- and post-conditions on program operations. For example, Figure 1 specifies the behavior of the program operations for \texttt{IntSet}, and Figure 2 describes the abstract values and trait functions for \texttt{IntSet}. Such model-oriented (or two-tiered) specifications make it easy to specify abstract data types incompletely. For example, the \texttt{choose} operation of \texttt{IntSet} is incompletely specified, because it can be implemented in several different ways. Incomplete specification is often good, since it allows one to leave design decisions open for either subtypes or implementations.

A common and important example of an incomplete supertype specification is a specification of a type that is missing some operations; e.g., operations that create objects. One might define a type \texttt{IntCollection} by giving a specification like \texttt{IntSet}, but without the operation \texttt{null}. Such \textit{virtual types} are useful as supertypes of more specific types (such as \texttt{IntSet}), and allow one to specify and verify programs at a high level of abstraction. Because one separately specifies the abstract values of a type and the program operations, one can describe the effect of the program operations precisely, even if there is no way to create an object of such a type in a program.

Meaning is given to sets of type specifications instead of to individual type specifications, since type specifications refer to other types (e.g., supertypes). Informally, the meaning of a set of type specifications is the set of program modules that implement the specifications (in some particular language). The exact notion of "implementation" is dependent on the programming language. In general, however, a module can be shown to implement a type specification by providing an abstraction relation \texttt{A}, that relates the objects created by the module (e.g., arrays) to their abstract values (e.g., mathematical sets), and by showing that operations satisfy the specified pre- and post-conditions. Since the pre- and post-conditions are stated using trait functions that apply to abstract values, one must use \texttt{A} to obtain abstract values and then check that the pre- and post-conditions are satisfied by each operation [1].

How can the details of abstraction relations and particular programming languages be ignored? One way is to provide a mathematical abstraction of implementations. For immutable types, an adequate abstraction is an algebraic model. An \textit{algebraic model} of a set of type specifications contains sets of abstract values, trait functions, and program operations. The abstract values (e.g., sets) are abstractions of object representations (e.g., arrays) that can be created by an implementation. The trait functions are functions on abstract values; these must satisfy their specification in the traits used by the set of type specifications.
The operations of an algebraic model are abstractions of the implementations of program operations on objects; each is a relation on abstract values that mimics the effect of the implementation's operation at the level of abstract values. (Relations are necessary to model nondeterminism, as in the \texttt{choose} operation.) In sum, an algebraic model is an abstract implementation that also contains an interpretation of the trait functions. Language details are suppressed by taking the meaning of a set of type specifications to be the set of abstract models that satisfy the type specifications.

A set of type specifications also determines a syntactic interface; this interface is used by programs to manipulate objects of the specified types. The syntactic interface is called a \textit{signature}, and contains the names of all the types, a binary relation on type names (the specified subtype relation), the names of program operations and trait functions, and a partial mapping, $ResType$, that gives the expected result type of calls to trait functions and program operations. $ResType$ takes a program operation or trait function name and a tuple of types and returns the expected result type (if any) for that operation. For example,

$$ResType(\texttt{ins}, \langle \texttt{Interval}, \texttt{Int} \rangle) = \texttt{IntSet}.$$ 

An algebraic model also has a signature. For simplicity, let the meaning of a set of type specifications with signature $\Sigma$ be a set of algebraic models with signature $\Sigma$.

### 5.1.2 Syntactic Constraints on Subtypes give Modular Specification

The reasoning method imposes the following constraints on signatures. First, if one can send a message such as \texttt{choose} to a supertype object, then one must also be able to send that message to a subtype object. This prevents surprises such as “message not understood.” Similarly, if a trait function name can be applied to a supertype's abstract values, then it should also apply to the abstract values of subtypes. So if $ResType(\texttt{isEmpty}, \langle \texttt{IntSet} \rangle)$ is defined, then $ResType(\texttt{isEmpty}, \langle \texttt{Interval} \rangle)$ must also be defined. Second, one must be able to interpret the expected result types given by $ResType$ as upper bounds, even when arguments types are lowered. For example, if $\leq$ is the specified subtype relation, then

$$ResType(\texttt{ins}, \langle \texttt{Interval}, \texttt{Int} \rangle) \leq ResType(\texttt{ins}, \langle \texttt{IntSet}, \texttt{Int} \rangle).$$

That is, $ResType$ must be monotonic in $\leq$ [8].

These constraints on signatures, although not sufficient to guarantee modular verification, guarantee modularity of specifications. Recall that modularity of specifications means that when one adds new types to a program, one need not respecify existing functions and types. Function and operation specifications are written as if the actual arguments had the
specified types and do not explicitly mention subtypes. An example is given in Figure 6. However, objects of subtypes of the specified types are allowed as arguments, which allows programmers to exploit subtype polymorphism. Such specifications are meaningful because the trait functions used in the specification can be applied to abstract values of the subtypes, by the above constraints. In effect the meaning of a specification such as Figure 6 is given by using dynamic overloading of the trait function names that appear in assertions. For example, if one knows that the abstract values of \( i v 1 \) and \( i v 2 \) are the intervals \([3, 27]\) and \([15, 73]\), then a description of the result of the call \( \text{inBoth}(i v 1, i v 2) \) can be obtained by substituting the abstract values of the actuals for the formals in the post-condition of \( \text{inBoth} \), obtaining the formula \((i \in [3, 27]) \& (i \in [15, 73])\), which is interpreted using the version of \( \in \) appropriate for the abstract values of intervals. Hence it is possible to discuss the testing and correctness of implementations of such specifications for all permitted arguments. Since subtypes are not mentioned explicitly in a function or operation specification, when a new subtype is added to the program, such a specification need not be changed.

5.1.3 Semantic Constraints on Subtypes give Modular Verification

Syntactic constraints are not enough to ensure sound, modular verification. The problem is illustrated in Figure 7, which illustrates static reasoning about the message-passing expression \( s . \text{choose}() \). Suppose that \( s \) is thought of as having nominal type \( \text{IntSet} \), as it would be before the type \( \text{Interval} \) was added to the program. To conclude that the value returned by \( \text{choose} \), called \( i \), satisfies the post-condition \( "i \in s" \) as specified for the type \( \text{IntSet} \), it would suffice to show that \( s \) satisfies the pre-condition \( "-(i \text{isEmpty}(s))" \). This reasoning would be adequate before the type \( \text{Interval} \) is added to the program. However, with the type \( \text{Interval} \) as a subtype of \( \text{IntSet} \), the identifier \( s : \text{IntSet} \) might denote an object \( s' \) of \( \text{IntSet}' \)’s subtype \( \text{Interval} \), instead of some object \( s \) of the type \( \text{IntSet} \). So at run-time the operation invoked is not the \( \text{choose} \) operation from the method dictionary associated with instances of \( \text{IntSet} \), written \( s . \text{choose} \) in the figure, but instead the operation \( s' . \text{choose} \).

The problem is that \( s' . \text{choose} \) might not satisfy the specification used during verification, since the \( \text{choose} \) operation of the type \( \text{Interval} \) has different pre- and post-conditions than the \( \text{choose} \) operation of \( \text{IntSet} \). Even if the pre- and post-conditions happened to be textually identical, the assertions might have different meanings for each type, since they rely on the meanings of trait functions such as \( \text{isEmpty} \)” that are interpreted differently for each type.

A solution is to require that there be a relationship, called a simulation, between the actual argument \( s' \) and the argument that was imagined during program verification (\( s \)).
\( \neg(\text{isEmpty}(s)) \quad s \xrightarrow{s\_\text{choose}} i \quad i \in s \)

\((\text{IntSet})\)

true \quad s' \xrightarrow{s'\_\text{choose}} i' \quad i = \text{leastElement}(s)

\((\text{Interval})\)

Figure 7: The problem with verification of the message-passing expression \( s\_\text{choose}() \). At the top is \text{IntSet}'s \text{choose} operation, at the bottom is \text{Interval}'s. At the left are these operation's pre-conditions, to the right are their post-conditions.

\( \neg(\text{isEmpty}(s)) \quad s \xrightarrow{s\_\text{choose}} i \quad i \in s \)

\((\text{IntSet})\)

\((\text{Interval})\)

Simulates-as \text{IntSet}

Simulates-as \text{Int}

true \quad s' \xrightarrow{s'\_\text{choose}} i' \quad i = \text{leastElement}(s)

Figure 8: Simulation solves the verification problem. This commutative diagram illustrates how simulation relationships are preserved by the program operation \text{choose}.

An example of a simulation relationship is specified following the keyword \textbf{by} in Figure 3. By the definition of the trait function “toSet”, each \text{Interval} with abstract value \([i, j]\) simulates an \text{IntSet} with abstract value \([i, i + 1, \ldots, j - 1, j]\).

Informally, the properties of a simulation relation are just those needed to make verification work, by connecting the pre- and post-conditions of the supertype and the subtype. The necessary conditions can be seen in Figure 8. One must show that “\( \neg(\text{isEmpty}(s')) \)” implies “\( \neg(\text{isEmpty}(s)) \)”, so that the \( s \) that the verifier imagined satisfies the pre-condition whenever \( s' \) does. This leads to the condition that simulation preserves the truth of assertions. (The assertion “\( \neg(\text{isEmpty}(s')) \)” makes sense, because the trait function “\( \text{isEmpty} \)” is defined for the abstract values of \text{Interval} as well as \text{IntSet}.) One must also show that each possible result \( i' \) simulates one of the possible results \( i \) that the verifier imagined. This leads to the condition that simulation is preserved by message passing.

To guarantee the preservation of simulation by message passing and assertions, it is enough to require that simulation relationships be preserved by program operations and by trait functions. This property of simulation relations is called the \textit{substitution property} (as in algebraic homomorphisms). For example, if \( q \) denotes the \text{Interval} \([1, 3]\) and \( r \) the
**IntSet** \{1, 2, 3\}, then \( q \) simulates \( r \). Thus by the substitution property:

\[
\begin{align*}
q.size() & \text{ simulates } r.size() \\
q.ins(0) & \text{ simulates } r.ins(0) \\
2 \in q & \text{ simulates } 2 \in r \\
\text{isEmpty}(q) & \text{ simulates } \text{isEmpty}(r).
\end{align*}
\]

For nondeterministic operations, such as \texttt{choose}, each possible result of \( q.\texttt{choose()} \) must simulate some possible result of \( r.\texttt{choose()} \). Simulation is not symmetric, since \( r.\texttt{choose()} \) may have more possible results than \( q.\texttt{choose()} \). Besides the substitution property, a simulation relation must be such that every object of a subtype simulates some object of each of its supertypes. Other properties required of simulation relations are described below.

Formally, simulation relations are families of binary relations, one per type, among abstract values. Each relation "simulates-as-T" relates the abstract values of subtypes of \( T \). For example, the following lists all the relationships between \{1, 3\} (an \texttt{Interval}) and \{1, 2, 3\} (an \texttt{IntSet}):

\[
\begin{align*}
[1, 3] & \text{ simulates-as-IntSet } \{1, 2, 3\} \\
[1, 3] & \text{ simulates-as-IntSet } \{1, 3\} \\
[1, 3] & \text{ simulates-as-Interval } \{1, 3\} \\
\{1, 2, 3\} & \text{ simulates-as-IntSet } \{1, 2, 3\}.
\end{align*}
\]

It is \emph{not} true that \{1, 2, 3\} simulates-as-\texttt{IntSet} \{1, 3\}, because the 2 is a possible result of \texttt{choose} on \{1, 2, 3\}, but it is not a possible result of \texttt{choose} on \{1, 3\}. The relation "simulates-as-\texttt{Interval}" is not defined on the abstract values of type \texttt{IntSet}, because \texttt{IntSet} is not specified to be a subtype of \texttt{Interval}.

The substitution property is formally defined by requiring that results are related at the expected result type, defined using \texttt{ResType}. For example,

\[
\begin{align*}
[1, 3] & \text{ simulates-as-IntSet } \{1, 2, 3\} \\
4 & \text{ simulates-as-Int } 4
\end{align*}
\]

so the substitution property says that the program operation \texttt{ins} must preserve the simulation:

\[
[1, 3].\texttt{ins}(4) \text{ simulates-as-IntSet } \{1, 2, 3\}.\texttt{ins}(4),
\]

where the results are related at the type \texttt{IntSet} because

\[
\texttt{ResType(\texttt{ins}, <\texttt{IntSet, Int}>)} = \texttt{IntSet}.
\]
The tuple \((\text{IntSet}, \text{Int})\) is used as the second argument to \(\text{ResType}\) above, because the actual arguments to \(\text{ins}\) were related at those types.

Besides the substitution property, to be a simulation a family of relations must satisfy four additional properties. First, each abstract value of a subtype must simulate some abstract value of each of its supertypes. Formally, this means that if \(S \leq T\), then for each \(q\) that is an abstract value of \(S\), there is some \(r\) that is an abstract value of \(T\) such that \(q\) simulates-as-\(T\) \(r\). Thus the simulation view is that of the supertype. The second property allows one to view an object as having a supertype of its exact type without invalidating one’s knowledge about that object at a subtype. This is formalized by the following condition: if \(S\) is specified to be a subtype of \(T\) and \(q\) simulates-as-\(S\) \(r\), then \(q\) simulates-as-\(T\) \(r\). The third property ensures that an abstract value that has no information content cannot simulate anything else. That is, if one considers nontermination (⊥) as an abstract value, then it can only simulate itself. The fourth property ensures that simulation agrees with external observations of programs. External observers can only see outputs of objects of types that are built-in to the programming language (e.g., \(\text{Bool}\) and \(\text{Int}\)). So the fourth property requires that for each such built-in type \(V\), simulation-as-\(V\) must be equality. For example, “true” cannot simulate-as-\(\text{Bool}\) “false”.

Simulation plays a central role in defining the semantic constraints on the specified subtype relation. For example, the constraints on the subtype relationship between \(\text{Interval}\) and \(\text{IntSet}\) can be informally summarized as follows: for each implementation of \(\text{Interval}\) there must be some implementation of \(\text{IntSet}\) such that each \(\text{Interval}\) object simulates some \(\text{IntSet}\) object in that implementation (where “simulates” means simulates-as-\(\text{IntSet}\)). Why must there exist “some” implementation of \(\text{IntSet}\) with this property? In a given program the implementation of \(\text{IntSet}\)’s \(\text{choose}\) operation might return the greatest element, so no simulation would be possible between the implementations in that program. (Recall that \(\text{Interval}\)’s \(\text{choose}\) operation always returns the least element of an interval.) However, during verification one uses properties of the \textit{specification} of \(\text{IntSet}\), not properties of a particular implementation. The specification allows the least element to be returned. Thus to show that the specified subtype relation meets the semantic constraints, one must, in general, use a different implementation of \(\text{IntSet}\) than the one in the given program.

The above informal idea breaks down for virtual types, types that have no class operations, since there will be no objects of such types in a program. Thus one cannot ask whether an object simulates an object of a virtual type, since there are no such objects to simulate. The use of algebraic models of type specifications avoids this difficulty, because
even virtual types have abstract values. For a virtual type specified in the Larch style, these abstract values are specified in the based on clause of a type specification. For example, let the virtual type IntCollection be specified like IntSet but without its null operation. The abstract values of IntCollection would then be mathematical sets generated by the trait IntSetTrait. One could form a simulation relation that shows how an abstract values of type Interval simulates an abstract value of type IntCollection. Thus the use of algebraic models allows one to treat virtual types and “normal” types in the same way.

So the formal semantic constraints on the specified subtype relation are as follows: for each algebraic model of the set of type specifications, there must exist some algebraic model such that there is a simulation relation between the first model and the second. Here an “algebraic model” plays the role of an “implementation” in the informal discussion above.

To see how the semantic constraints on the specified subtype relation aid modular verification, consider Figure 8 again. At run-time the object sent the choose message has an Interval abstract value s'; imagine that this abstract value s' is from an algebraic model C. If the specified subtype relation satisfies the above conditions, then there is an algebraic model, A, such that there is a simulation between C and A. By definition of a simulation, there must be some s in A such that s' simulates-as-IntSet s and s has type IntSet. Thus one can always find the hypothetical s in the figure.

There are two minor differences in verification with subtypes as opposed to conventional verification. In conventional verification the “rule of consequence” allows one to use a stronger pre-condition to conclude a weaker post-condition than would be necessary when calling a program operation. For example, “size(s) = 2” implies “¬isEmpty(s)”, so if one knows that the value of s satisfies the former, then it satisfies the pre-condition of choose for IntSet. However, implication is tricky when subtypes are present. For example, the assertion

\[((\text{size(s) = 1} \& (3 \in s)) \Rightarrow (s = \{3\})\]

is valid when “s” denotes an IntSet, but is not valid when “s” denotes an Interval, since the abstract value of an Interval would have the form [3, 3], not \{3\}. Thus for sound verification with subtypes, one cannot use equality (\(=\)) of abstract values with the rule of consequence or in pre- and post-conditions, except for built-in types such as Bool and Int that are assumed not to have subtypes [7].

The second difference from conventional verification occurs when one explicitly exploits subtyping. Figure 5 shows an example, where in the main program some arguments to doSomething have nominal type Interval. If one has an assertion that characterizes the value of such an object bound to an identifier iv: Interval, then one must translate that to
an assertion that characterizes the abstract value at the supertype; that is, when the object is bound to \( s_1 : \text{IntSet} \) or \( s_2 : \text{IntSet} \) by the call of \texttt{doSomething}. Since the meaning of an assertion is given by dynamic overloading of trait functions, one might substitute \( s_2 \) for \( iv \) throughout the assertion, except that the assertion might no longer type check. For example, in the assertion “\( \text{leastElement}(iv) = 3 \)” it does not make sense to substitute \( s_2 \) for \( iv \), as “\( \text{leastElement} \)” is not a trait function that applies to \( \text{IntSet}s \). So in general, one must use the rule of consequence to weaken the assertion to a form that only uses trait functions defined on the supertype. For example, one might weaken “\( \text{leastElement}(iv) = 3 \)” to “\( 3 \in iv \)”. A verification rule then allows one to conclude that “\( 3 \in s_2 \)” holds, in contexts where \( iv \) is assigned to \( s_2 \) or passed as the actual argument to the formal \( s_2 \). This is sound because in such contexts \( s_2 \) denotes the same object as \( iv \).

6 Discussion

6.1 Formal Ideas Guide Informal Reasoning

The programming method described above corresponds to informal techniques used by object-oriented programmers. The key idea is that objects of a subtype must “behave like” objects of that type’s supertypes. The notion of “behaves like” for objects of immutable types has been formalized as simulation above. But the formalization itself is not the most important lesson. More important is the end achieved by subtyping: modular specification and verification.

The goal of modular reasoning can also be used to guide both programmers and researchers who need a precise concept of subtyping to reason about programs that fall outside the limitations of the formal techniques presented above. That is, the concept of subtyping must be strong enough to permit modular specification and verification. For example, when reasoning about concurrency, checking that type \( S \) is a subtype of \( T \) would also involve checking that the use of objects of type \( S \) does not invalidate any assumptions made about absence of deadlocks that one could derive from the specification of type \( T \). A pressing research problem is how to independently describe such notions of subtyping. A related research problem is how to formally state and verify subtype relationships.

Because the semantic requirements on subtype relationships are so strong, it is necessary for designers to design new types with subtyping in mind. This is another facet of the idea of designing a program and its correctness argument at the same time. To guide the design process, one can use the idea of a “simulation relation” to ensure that the new type will be a subtype of the desired existing types. Since most object-oriented designs will involve types that are beyond the limitations of the formal definition of simulation, this guidance
will only be heuristic. But one can use the informal idea of subtyping to check (informally) that the desired subtype relationships are achieved.

6.2 Implications for Language Design

Subtyping and subclassing are distinct concepts that can and should exist separately. A programming language should allow one to use inheritance for shorthand definition of classes, regardless of subtyping relationships. Furthermore, one should be able to define and specify subtypes regardless of whether they are implemented with subclasses. This is one point of the Interval example. The type Interval is specified as a subtype of IntSet but a class Interval would not be defined as a subclass of a class IntSet (as the data structure would be inefficient).

In Smalltalk-80, there is no notion of type checking based on subtype relationships; hence programmers can use inheritance freely, but must enforce a disciplined use of subtypes by themselves.

A language can aid the disciplined use of subtypes in specification and verification if its type system allows one to declare subtype relationships and if it ensures that each expression can only denote objects whose type at run-time is a subtype of the expression’s nominal type. For example, the type system in C++ is barely adequate (if one ignores casts, and other obvious insecures), since one can declare a subclass relationship to be protected or private as opposed to public, and the C++ type system only considers public subclasses to be subtypes. So in C++ one can make subclasses that do not implement subtypes, and the type system will not allow pointers to objects of such subclasses to be used where pointers to objects of their superclasses are expected.

In C++ one cannot implement a subtype except as a subclass of the classes that implement the type’s supertypes. But that would force one to use an inefficient representation for Interval inherited from IntSet. To avoid the inefficient representation one would use a virtual class IntCollection, and implement subclasses Interval and IntSet. The class IntCollection would not allow one to create objects, would only have virtual operations, and would not define a representation for objects (instance variables). The representation and operations would be defined by the (public) subclasses IntSet and Interval. However, this plan requires forethought; if one has not planned for the type Interval during design, then one is unlikely to define the class IntCollection. So one will be obliged to make changes to other code when the type Interval is added to the program, if only to change some occurrences of IntSet as an argument type to IntCollection.
7 Conclusions

Modular specification of object-oriented designs and modular verification of object-oriented programs are important problems. Key ideas for solving these problems are behavioral subtype relationships and the use of supertypes to “stand for” subtypes during specification and verification. To ensure soundness of verification, the specified subtype relation must satisfy certain semantic constraints, and the nominal type of each expression must be a supertype of the types of the objects it may denote. A key semantic constraint is that each object of the subtype should simulate some object of each of its supertypes, and that this simulation should be preserved by message passing and assertions.

Acknowledgements

Thanks to Bill Weihl for guiding the research described above, and to Bill Weihl, Kelvin Nilsen, Al Baker, and Dhara Krishna for comments on drafts. Thanks to the anonymous referees for helping to clarify my thinking.

References


Related Work (Sidebar)

Gary T. Leavens
Department of Computer Science
Iowa State University, Ames, Iowa 50011 USA
leavens@cs.iastate.edu (515) 294-1580
February 10, 1991

The standard informal definition of subtype relationships is that each object of the subtype must “behave like” some object of the supertype [1] [2].

Liskov has described how subtype relationships can be used during design to record decisions that refine type specifications, to localize the effects of changes to type specifications, and to group and classify types [3]. LaLonde also uses subtype relationships as a means of classifying types by behavior [4]. Neither Liskov nor LaLonde gives a formal definition of subtype relationships.

Some semi-formal specification and verification techniques appear in Meyer’s book on Eiffel [5]. In chapter 11, Meyer states that a subclass should be designed to implement a subtype. The “assertion redefinition rule” states that if \( r \) is an operation of a class \( A \) and \( B \) is a subclass of \( A \), then the pre-condition of \( r \) in the specification of \( B \) may be no stronger than the pre-condition of \( r \) in \( A \), and the post-condition of \( r \) in the specification of \( B \) must be no weaker than the post-condition of \( r \) in \( A \) [5, Page 256]. This rule ensures that the implementation of an operation in a subclass (\( B \), satisfies the specification of that operation in the superclass (\( A \)).

Reynolds has studied partial orders on types in the setting of his category sorted algebras [6]. The semantic requirement that Reynolds imposes on the subtype relation is illustrated by the following example. Suppose \texttt{Integer} is a subtype of \texttt{Float}, \( a \) and \( b \) are objects of type \texttt{Integer}, and \texttt{to\_Float} is the coercion function from \texttt{Integer} to \texttt{Float}. The coercion function \texttt{to\_Float} must satisfy the \textit{substitution property}:

\[
to\_Float(a + b) = to\_Float(a) + to\_Float(b)
\]

where the “+” on the left is \texttt{Integer} addition and the “+” on the right is \texttt{Float} addition. Requiring that the coercion satisfies the substitution property with respect to operations...
such as "+" ensures that one can reason about overloading and coercion without an exhaustive case analysis. A similar idea is found in the work of Bruce and Wegner [7].

P. America has independently developed a definition of subtype relationships [8]. Like Meyer, America's definition is based on implications between pre- and post-conditions of operations. However, unlike Meyer, America does not use program operations in assertions. Instead, types are specified by describing the abstract values of their instances, and the post-condition of each program operation relates the abstract values of the arguments to the abstract value of the result. The set of abstract values of a subtype may be described differently than the set of abstract values of a supertype. Thus, for a subtype relationship, America requires a "transfer function", \( f \), that maps the abstract values of the subtype to the abstract values of the supertype. Furthermore, for each instance operation of the supertype, it is required that

\[
\text{Pre}(\text{Super}) \circ f \Rightarrow \text{Pre}(\text{Sub})
\]

\[
\text{Post}(\text{Sub}) \Rightarrow \text{Post}(\text{Super}) \circ f
\]

where the transfer function \( f \) is used to translate assertions of the supertype so that they apply to the abstract values of the subtype. In practice, the above requirements often mean that the transfer function must have a substitution property with respect to the program operations. However, the types that America specifies do not have class operations, hence his notion of subtyping is identical to the notion of refinement.

The main line of type theoretic research on subtyping has been carried on by Luca Cardelli. His landmark paper "A Semantics of Multiple Inheritance" [9] showed the soundness of subtyping rules for function types, immutable records, and immutable variants. But neither this paper nor more sophisticated systems (such as [10]) give subtype rules for abstract data types in general. That is, such type systems do not give general rules that can say whether \texttt{Interval} is a subtype of \texttt{IntSet} based on their specifications.

References


