Reasoning About Object-Oriented Programs That Use Subtypes

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/92

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Reasoning About Object-Oriented Programs That Use Subtypes

Abstract
Programmers informally reason about object-oriented programs by using subtype relationships to classify the behavior of objects of different types and by letting supertypes stand for all their subtypes. We describe formal specification and verification techniques for such programs that mimic these informal ideas. Our techniques are modular and extend standard techniques for reasoning about programs that use abstract data types. Semantic restrictions on subtype relationships guarantee the soundness of these techniques.

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

Disciplines
Systems Architecture
Reasoning About Object-Oriented Programs That Use Subtypes

TR90-03B
Gary T. Leavens

March 1990
Revised
Reasoning about Object-Oriented Programs that use Subtypes
(Extended Abstract)

Gary T. Leavens and William E. Weihl

TR #90-03b
March, 1990
(Revised July, 1990)

This paper is a revision of TR #90-03. It will appear in the ECOOP/OOPSLA '90 Proceedings.

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. © 1990 ACM

Department of Computer Science
Iowa State University
Ames, Iowa 50011-1040, USA
Reasoning about Object-Oriented Programs that use Subtypes

(Extended Abstract)

Gary T. Leavens*
Department of Computer Science
Iowa State University, Ames, Iowa 50011 USA
leavens@atanasoff.cs.iastate.edu

William E. Weihl
Laboratory for Computer Science
Massachusetts Institute of Technology, Cambridge, Mass. 02139 USA
weihl@lcs.mit.edu

Abstract

Programmers informally reason about object-oriented programs by using subtype relationships to classify the behavior of objects of different types and by letting supertypes stand for all their subtypes. We describe formal specification and verification techniques for such programs that mimic these informal ideas. Our techniques are modular and extend standard techniques for reasoning about programs that use abstract data types. Semantic restrictions on subtype relationships guarantee the soundness of these techniques.

1 Introduction

The message-passing mechanism of an object-oriented language such as Smalltalk-80 [GR83] allows one to write polymorphic code; i.e., code that works for objects of many types. However, reasoning about programs that use message-passing is difficult because there may be many different operations that could be executed by a message send. Furthermore, the same piece of code may call different operations during different executions.

To obtain the advantage of extensibility promised by object-oriented methods, specification and verification techniques must be modular in the sense that when new types of objects are added to a program, unchanged program modules should not have to be respecified or reverified.

We present a modular specification and verification technique for reasoning about message-passing programs that is based on the concepts of subtype relationships and nominal type. Informally, the reasoning technique can be summarized as follows.

- One specifies the data types to be used in the program along with their subtype relationships.
- Functions are specified by describing their effects on actual 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 must be verified to ensure that they have the appropriate semantics. Intuitively, if a type S is a subtype of a type T, then every object of type S must behave like some object of type T.
- One associates with each expression in the program a type, called the expression's nominal type, with the property that an expression may only
denote objects having a type that is a subtype of that expression's nominal type. (These types may be introduced solely for program verification, or they may coincide with the types of the programming language.)

- Verification that a program meets its specification is then 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 key to the soundness of our method is the semantic requirements on subtype relationships [Lea89]. The method has been refined from [Lea89].

The rest of this paper is organized as follows. In Section 2 we describe the programming language used in this paper. Next, in Section 3 we present some background. In Section 4 we describe the problem in more detail. In Section 5 we present our method, and in Section 6 we discuss the soundness of our method. Finally, in Sections 7, 8, and 9, we discuss related work, future work, and some conclusions.

## 2 Programming Language

In this paper we use an applicative programming language that can observe objects of immutable types by message-passing. (An immutable type is an abstract type whose instances have no time-varying state.) This is a first step towards reasoning for more realistic languages. The language is an extension of the simply-typed, applicative-order lambda calculus; the syntax of the language is given in Figure 1. The syntax uses \texttt{fun} instead of \texttt{λ}, and a program is a function from input arguments to outputs. There is no syntax for implementing types (i.e., classes); in this paper we will focus solely on programs that use such types and the specifications of such types. Function identifiers in programs are written in a \texttt{slanted} font to distinguish them from message names written in \texttt{typewriter} font. Function identifiers are statically bound to functions; message names are dynamically bound as described below.

Type checking for this language is based on subtyping, using techniques from Reynolds's category sorted algebras [Rey81] [Rey85]. Each expression is statically assigned a nominal type, determined from the information given in type specifications and program declarations. The type specifications determine a partial function \texttt{ResType}, which maps message names and tuples of types to expected result types, and a user-specified reflexive and transitive relation among types, ≤, called the subtype relation.

\[
\begin{align*}
\langle \text{program} \rangle &::= \langle \text{prog} \ (\langle \text{decls} \rangle ) \ :\ \langle \text{type} \rangle = \langle \text{expr} \rangle \\
&\quad | \langle \text{rec fun def} \rangle \ \langle \text{program} \rangle \\
\langle \text{decls} \rangle &::= \langle \text{decl list} \rangle \ | \langle \text{empty} \rangle \\
\langle \text{decl list} \rangle &::= \langle \text{decl} \rangle \ | \langle \text{decl list} \rangle , \langle \text{decl} \rangle \\
\langle \text{decl} \rangle &::= \langle \text{identifier} \rangle : \langle \text{type} \rangle \\
\langle \text{empty} \rangle &::=\ \\
\langle \text{expr} \rangle &::= \langle \text{identifier} \rangle \\
&\quad | \langle \text{message name} \rangle (\langle \text{expr list} \rangle ) \\
&\quad | \langle \text{function identifier} \rangle (\langle \text{exprs} \rangle ) \\
&\quad | (\langle \text{function abstract} \rangle ) (\langle \text{exprs} \rangle ) \\
&\quad | \textit{if} \ \langle \text{expr} \rangle \ \textit{then} \ \langle \text{expr} \rangle \ \textit{else} \ \langle \text{expr} \rangle \ \textit{fi} \\
&\quad | (\langle \text{expr} \rangle ) \\
\langle \text{exprs} \rangle &::= \langle \text{expr list} \rangle \ | \langle \text{empty} \rangle \\
\langle \text{expr list} \rangle &::= \langle \text{exprs} \rangle \ | \langle \text{expr list} \rangle , \langle \text{expr} \rangle \\
\langle \text{function abstract} \rangle &::= \textit{fun} \ (\langle \text{decls} \rangle ) \ (\langle \text{expr} \rangle ) \\
\langle \text{rec fun def} \rangle &::= \textit{fun} \ \langle \text{function identifier} \rangle \\
&\quad (\langle \text{decl list} \rangle ) : \langle \text{type} \rangle = \langle \text{expr} \rangle ;
\end{align*}
\]

Figure 1: Programming language syntax.

For example, consider the message-passing expression \texttt{add(a,b)}. The nominal types of the arguments \texttt{a} and \texttt{b} are given in their declarations, say \texttt{Fraction} and \texttt{Integer}. The nominal type of the result is then \texttt{ResType(add,\{Fraction,\}Integer)}. To ensure that nominal types can be thought of as upper bounds and that operations of supertypes may be applied to subtypes, \texttt{ResType} must be monotone in the following sense: for all message names \texttt{g}, and for all tuples of types \texttt{S} ≤ \texttt{T}, if \texttt{ResType(g,\{T\})} is defined, then so is \texttt{ResType(g,\{S\})}, and \texttt{ResType(g,\{S\})} ≤ \texttt{ResType(g,\{T\})}. (This is a constraint on the types used in a program.) Arguments to functions are allowed to have types that are subtypes of the declared argument types.

## 3 Background

In this section we discuss subtype polymorphism, and how it differs from the polymorphism found in more conventional languages.
fun sqrt(x:Fraction): Fraction =
sqrtIter(1, x);

fun sqrtIter(guess, x:Fraction): Fraction =
if good(guess, x)
then guess
else sqrtIter(improve(guess, x), x) fi;

fun good(guess, x:Fraction): Boolean =
lt(abs(sub(x, mul(guess, guess))), create(Fraction, 1, 1000));

fun improve(guess, x:Fraction): Fraction =
mean(guess, div(x, guess));

fun mean(a, b:Fraction): Fraction =
div(add(a, b), 2);

Figure 2: Implementation of the function sqrt.

The polymorphism in Smalltalk-80 programs is a result of Smalltalk’s dynamic overloading of message names. Wadlak and Blott’s method dictionaries provide a good explanation of this style of message passing [WB89]. A method dictionary is a map from the names of overloaded operators to the operations specific to a given type. The method dictionaries are associated with objects in Smalltalk-80, and a message send is evaluated by consulting the receiving object’s method dictionary and invoking the operation with the given message name. Thus an operation in Smalltalk-80 can be polymorphic, since a call to it is implicitly passed the method dictionaries needed to manipulate objects of different types.

Our language has a more complex form of dynamic overloading, in which method dictionaries map message names to dispatchers, which are mappings from tuples of types to operations specific to a combination of argument types. As in the Common LISP Object System (CLOS) [Kee89], a dispatcher finds an operation based on the run-time types of all the actual arguments. For example, the function sqrt of Figure 2 (code adapted from [AS85, Page 22]) is polymorphic, because the implicit method dictionary passed to sqrt is defined for the message names lt, abs, etc. used at run-time and because the relevant dispatchers are defined on tuples of types made from Fraction and its subtypes, such as Integer. So sqrt may take both Fraction and Integer arguments.

Cardelli and Wegner have called the kind of polymorphism exhibited by the implementation of sqrt in Figure 2 “inclusion polymorphism” [CW85], although we prefer the term subtype polymorphism. Subtype polymorphism is distinguished by two features from other kinds of parametric polymorphism: the dynamic binding of operation names to operations based on the run-time types of their arguments, and the possibility that a given expression may denote objects with different types at run-time. With subtype polymorphism, it is impossible, in general, to statically determine the type of object a given expression will denote at run-time. For example, consider the program of Figure 3.

When evaluating the program’s body, the formal parameter of sqrt may denote an object either of type Integer (i.e., 16) or of type Fraction (i.e., 3/4), depending on the program’s input. There may be different implementations of the operations add, etc. for each combination of argument types. This makes it difficult to reason about a program that uses subtype polymorphism.

4 The Problem

Our goal is to obtain a modular specification and verification method for programs that use message-passing and subtype polymorphism. Even if formal verification of such programs is not practical, the desire for modularity in large programs makes it important to give careful informal specifications of functions and to reason informally about their use. A better understanding of formal techniques for specification and verification can serve as a guide to such informal reasoning.

An obvious approach is to adapt traditional reasoning techniques. For example, the traditional, parameterized specification of sqrt would have as parameters a type T, an object x of type T, and functions 1t, abs, sub, mul, and div that would allow the square root to be computed. (See, for example, [Gut80, Page 21], [Win83, Section 4.2.3], and [Gog84, Page 537].) The functions 1t, abs, etc. can be grouped into a single parameter: a method dictionary. It is necessary to specify the behavior of the functions in this method dictionary, since otherwise one cannot prove that the implementation of sqrt is correct. The problem with this approach is that to use such a specification during verification, the actual method dictionary must be known statically, so that one can verify its behavior.
fun sqrt(x: Fraction) returns(r:Fraction)
  requires 0 ≤ x
  ensures (0 ≤ r) & ((r * r) - x) ≤ (1/100))

Figure 4: Specification of the function sqrt.

However, for a language like Smalltalk-80, the method dictionary cannot, in general, be determined statically during verification of a call such as the one in Figure 3. For our language or CLOS the method dictionary has dispatchers for all combinations of argument types. So the use of traditional reasoning techniques leads to an exhaustive case analysis that must be repeated when new subtypes are introduced. In other words, this approach does not allow modular verification.

5 Overview and Example of the Method

Our approach extends traditional specification and verification techniques to cope with subtype polymorphism in a modular fashion. We first discuss specification of functions and abstract types, and then program verification.

5.1 Function Specifications

Our function specification technique is illustrated by the specification of sqrt given in Figure 4. To ensure modularity, the behavior of sqrt is described explicitly only for fixed types of arguments and results; that is, for the nominal types of the forms. But this specification is implicitly polymorphic, since the actual arguments passed to a call of sqrt may have types that are subtypes of the corresponding nominal argument types. For example, sqrt may take Integer arguments, since Integer is a subtype of Fraction.

The ensures clause (i.e., the post-condition) of sqrt in Figure 4 states how the value of the result is related to the values of the argument, assuming that it is of type Fraction. The requires clause describes the pre-condition of sqrt. Such a specification is a two-tiered [Win87] or abstract-model style [BJ82] specification. In such specifications, the characteristics, or abstract values, of objects are described mathematically, and the vocabulary of abstract values is used to specify functions and the operations of abstract types. Following Wing we describe the abstract values of types using Larch traits [GH86b]. The symbols "≤", "⋅", "", and "" used in the pre- and post-condition are the names of trait functions and are described in

the trait IntAndRat (Figure 5). Trait functions can be used in assertions but not in programs.

In the trait IntAndRat, the included traits Integer and Rational are found in [GH86a]. The names and signatures of additional trait functions are described after the keyword introduces. The constrains section is an equational specification of the trait functions. The terms in the exempts section are undefined.

For a function specification to be meaningful when the arguments have a subtype of their specified types, the specifier of a subtype must ensure that the trait functions used to describe the abstract values of a supertype can also be applied to the abstract values of each of its subtypes. In essence, the meaning of a specification is given by dynamic overloading for trait functions\(^1\) (similar to message-passing). For example, consider the call sqrt(16), in which the abstract value of the argument is 16. Because of the overloaded trait functions, a description of the result is obtained by substituting 16 for x in the post-condition, obtaining the formula "(0 ≤ x) & ((x * x) - 16) ≤ (1/100)". Hence the value of the result x must be non-negative and sufficiently close to 4. Since the trait functions apply to subtypes, the resulting formula describes the result equally well, whether it is a Fraction or an Integer. Similarly, the pre-condition is meaningful for arguments of type Integer as well as arguments of type Fraction.

An implementation of sqrt satisfies its specification if, whenever the arguments satisfy the pre-condition, it always terminates and the value of the result, when substituted for the formal result identifier (r), satisfies the post-condition.

5.2 Type Specifications

Type specifications describe the behavior of each type used in a program and also specify subtype relationships. The specification of a subtype relationship involves stating how each object of the subtype simulates the objects of its supertypes.

The specifications of the types Fraction and Integer appear in Figures 6 and 7, respectively. The specification of a type has a header followed by specifications for each of the operations provided by the type. The operation specifications are read like function specifications.

In the header of a type specification the operations are divided into class and instance operations; class operations are typically used to create new instances of a type, and instance operations are called by sending

\(^1\) The meaning of a specification is not given by coercing the abstract values of arguments, as in [Lee88].
IntAndRat: trait
   includes Integer,
   Rational with [rat1 for 1, rat0 for 0]
   introduces #/#: Int,Int -> R
   gcd: Int,Int -> Int
   [|]: R -> R
   numerator, denominator: R -> Int
   [|], numerator, denominator: Int -> Int
   #+#, #-, */#, #/#: R,Int -> R
   #==#: Int,Int -> Bool
   #==#, #<=#, #>=#, #<#, #<#: R,Int,Int -> R,
   #/#: Int,Int -> R
so that for all [n,m,d: Int, f,g,h: R]
   (1/1) = rat1
   (n/d) * (d/n) = rat1
   ((n+m)/d) = (n/d) + (m/d)
   {n} = if n<0 then -n else n fi
   {f} = if f<0 then -f else f fi
   gcd(n,m) = gcd(m,n)
   gcd(n,m) = gcd(-n,m)
   gcd(n,0) = |n|
   gcd(n*m, m*n) = gcd(n,m)*d
   ((numerator(f) = n)
     & (denominator(f) = d))
   = ((n/d) = f) & (d > 0)
     & (gcd(n,d) = 1))
   numerator(n) = n
   denominator(n) = 1
   (f == g) =
   ((numerator(f) = numerator(g)) &
    (denominator(f) = denominator(g)))
   (n == m) = (n = m)
   (f == n) = (f == (n/1))
   (n == f) = ((n/1) == f)
   f + n = f + (n/1)
   n + f = (n/1) + f
   % and so on for -, * , /, etc.
exempts for all [n,m: Int] n/0, n/(0/m),
   (0/m)/0

Fraction immutable type
   class ops create
   instance ops num, denom, add, sub, mul, div,
   abs, lt, equal

   based on sort R from trait IntAndRat
   op create(c: FractionClass, n,d: Integer)
     returns(f: Fraction)
     requires ~(d = 0)
     ensures f == n/d
   op num(f: Fraction) returns(i: Integer)
     requires i = numerator(f)
   op denom(f: Fraction) returns(i: Integer)
     ensures i = denominator(f)
   op add(f1,f2: Fraction) returns(f: Fraction)
     ensures f == (f1 + f2)
   op sub(f1,f2: Fraction) returns(f: Fraction)
     ensures f == (f1 - f2)
   op mul(f1,f2: Fraction) returns(f: Fraction)
     ensures f == (f1 * f2)
   op div(f1,f2: Fraction) returns(f: Fraction)
     requires ~(f2 == 0/1)
     ensures f == (f1 / f2)
   op abs(f: Fraction) returns(g: Fraction)
     ensures g == |f|
   op lt(f1,f2: Fraction) returns(b: Boolean)
     ensures b = (f1 < f2)
   op equal(f1,f2: Fraction) returns(b: Boolean)
     ensures b = (f1 == f2)

Figure 6: Specification of the type Fraction.

messages to instances. The header of a type’s specification includes two additional clauses: a based on clause, and an optional subtype of clause. The based on clause describes the abstract values of the objects of the type, by naming a sort and a Larch trait that specifies that sort. The abstract values of objects of type Fraction are elements of the sort R, which is taken from the trait IntAndRat. The trait IntAndRat, which is described in Figure 5, relates the included traits Integer and Rational by an additional infix trait function / that takes two integers and returns a fraction. The trait IntAndRat also specifies mixed mode trait functions; these are necessary so that the specification of a binary operation says what happens when only one argument is an object of a subtype. It is hoped that in the future the mixed mode trait functions can be specified more succinctly, perhaps by using order-sorted algebra [GM87]. One can always define them by first coercing all arguments to the supertype.

The optional subtype of clauses describe a relation
Integer immutable type

subtype of Fraction by n simulates n/1

class ops one

instance ops num, denom, add, sub, mul, div,
abs, lt, equal

based on sort Int from trait IntAndRat

op one(cIntegerClass) returns(i: Integer)
ensures i = 1

op num(i: Integer) returns(j: Integer)
ensures j = i

op denom(i: Integer) returns(j: Integer)
ensures j = 1

op add(i1,i2: Integer) returns(i: Integer)
ensures i = (i1 + i2)

op sub(i1,i2: Integer) returns(i: Integer)
ensures i = (i1 - i2)

op mul(i1,i2: Integer) returns(i: Integer)
ensures i = (i1 * i2)

op div(i1,i2: Integer) returns(f: Fraction)
requires −(i2 = 0)
ensures f == (i1 / i2)

op abs(i: Integer) returns(j: Integer)
ensures j = |i|

op lt(i1,i2: Integer) returns(b:Boolean)
ensures b = (i1 < i2)

op equal(i1,i2: Integer) returns(b:Boolean)
ensures b = (i1 = i2)

\( \leq \) among type symbols (the subtype relation), and a family of relations \( R \) between the abstract values of types (the simulation relation). For each supertype listed, one specifies for each object \( x \) of the subtype at least one object of the supertype that \( x \) "simulates." For example, the specification of the type \textbf{Integer} states that \textbf{Integer} is a subtype of \textbf{Fraction}, and that an integer with value \( n \) simulates a fraction with value \( n/1 \).

Formally, the relation \( \leq \) is the reflexive, transitive closure of the \textbf{subtype} of relationships given in the type specifications.

There is a relation \( R_T \) for each type \( T \). The relation \( R_T \) says how the abstract values of objects of each type \( S \leq T \) are to be viewed as objects of type \( T \). For example, for each integer value \( n \), \( n \in R_\text{Fraction} n/1 \), as specified in \textbf{Integer}’s \textbf{subtype} of clause. By convention, the following additional relationships are implicit in such specifications. For each type \( T \), the relation \( R_T \) includes the identity relation on the abstract values of objects of type \( T \) and all relations \( R_S \) such that \( S \leq T \); for example, the fraction \( n/d \) is related by \( R_\text{Fraction} \) to itself and \( R_\text{Fraction} \) relates the integer \( n \) to itself. Furthermore, the relationships compose transitively in the following sense: if \( S \leq T \) and \( a R_S b R_T c \), then \( a R_T c \).

The family \( R \) is used to verify that \( \leq \) has the necessary semantic properties to be a subtype relation. The relation \( \leq \) can also be viewed as summarizing information about \( R \). That is, if \( S \leq T \), then it is required that for every object of type \( S \), its abstract value is related by \( R_T \) to the abstract value of some object of type \( T \), and that \( R \) has the semantic properties described below. (In the programming language the relation \( \leq \) is also used by the type-checker.)

The binary operations provided by the type \textbf{Integer} have \textbf{Integer} as the type of their second argument, and most have \textbf{Integer} as the type of their result. Thus, for example, if \( a \) and \( b \) denote objects of type \textbf{Integer}, then \( \text{add}(a,b) \) must denote an \textbf{Integer}. The operation specification that determines the behavior of an invocation of \texttt{add} is the most specific specification whose argument types are supertypes of the types of the actual arguments, because message-passing is used at runtime. For example, the result of \texttt{add}(a,b) need only satisfy the specification of the add operation of \textbf{Integer} if \( a \) and \( b \) denote \textbf{Integers}. On the other hand, if \( a \) denotes a \textbf{Fraction}, then the result of \texttt{add}(a,b) is determined by the specification of the add operation of \textbf{Fraction}. (The semantic restrictions on subtype relationships ensure that these behaviors are related.) Such specifications would be well suited for the specification of CLOS programs [Kee89], where generic operations can be defined for various combina-
tions of argument types.

The pre- and post-conditions of operations must not use equality (=), except between terms of visible type — built-in types for which no subtypes are allowed, such as Boolean and Integer. Assertions that satisfy this condition are called subtype-constraining. Technically, this restriction is needed to ensure the soundness of program verification. However, the restriction is also intuitively necessary. Consider the pre-condition of the div operation in Figure 6. If the pre-condition were “(f2 = 0/1)” instead of “(f2 == 0/1)”, then it would be satisfied when f2 denoted the Integer 0, since the abstract value 0 is not the same as 0/1; this is probably not what the specifier meant. The trait function “==” does not test equality of abstract values; thus “0 == (0/1)” is true, because “numerator(0) = 0” and “denominator(0) = 1”, as specified in the trait IntAndRat. Equality (=) is not a trait function and cannot be redefined by subtypes. As another example, if the post-condition of div had been stated as “f = (fl / f2)”, then the abstract value of div(div(1,2),div(1,4)) would have to be 2/1 (a Fraction), not 2 (an Integer).

Inheritance of specifications by a subtype specification would be a useful extension to a practical specification language. For example, the specifications of the Integer operations num, denom and div are quite similar to their specification for Fraction arguments and could perhaps be inherited. One could then specify a subtype by specifying only the subtype’s class operations and those instance operations that are added by the subtype or that need to be further constrained.

5.3 Verification

Our approach to modular verification is to allow one to reason about expressions based on nominal type information.

Subtyping does not enter into the verification of a program directly. The only interaction is that the specified relation ≤ must be verified to have certain properties (see Section 6 below) and the type system must ensure that each expression can only denote objects whose type is a subtype of the expression’s nominal type. This separation is achieved by ensuring that the trait functions used to describe the abstract values of a supertype also apply to subtypes (with the appropriate semantics).

Although our language is applicable, we use a Hoare logic for program verification, because we are ultimately interested in verification of imperative programs.

Hoare-triples are written P {v = E} Q and consist of a pre-condition P, a result identifier v, an expression E, and a post-condition Q. (The name of the result identifier can be chosen at will, but cannot occur free in the pre-condition.) In an applicative language, expressions have results but do not change the environment in which they execute. So the post-condition describes the environment that results from binding the result identifier (v), which has a nominal type that is a supertype of E’s nominal type, to E’s value. Intuitively, P {v = E} Q is true if whenever P holds, then the execution of E terminates, and the value of E satisfies Q.

To simplify the verification system, the following rule is used to verify a message-passing expression or function call that has general expressions as arguments.

\[
\frac{\vdash P \{ \text{true} \} \quad \vdash P \{ y = g(E) \} \} Q}{\vdash P \{ \text{true} \} \quad \vdash P \{ y = g(E) \} \} Q}
\]

That is, to prove the desired triple (on the bottom) holds, one must show that the post-condition Q follows when the actual argument expressions are replaced by identifiers bound to the expressions’ values by a function abstract. The names and types of these identifiers must be chosen so an appropriate axiom for the inner message send or function call will apply, and so that the application on the top type-checks. For example, to prove the following triple

\[
\text{true} \{ f = \text{add}(3,4) \} \quad \vdash f = 7
\]

where f has nominal type Fraction, it suffices to prove the following triple (with the parts displayed vertically).

\[
\text{true} \{ f = (\text{add}(f1,f2)) (3,4) \} \quad \vdash f = 7
\]

With the above rule, the specifications of each type’s operations and each function specification can be taken as simple axioms. For example, there are two axioms for the message add:

\[
\vdash \text{true} \{ f = \text{add}(f1,f2) \} \quad \vdash f = (f1 + f2) \quad (4)
\]

\[
\vdash \text{true} \{ i = \text{add}(i1,i2) \} \quad i = (i1 + i2) \quad (5)
\]

The specification of sqrt generates the following axiom:

\[
0 \leq x \quad \vdash \{ x = \text{sqrt}(x) \} \quad (0 \leq x \land (\sqrt{x^2} - x) \leq (1/100)) \quad (6)
\]

These axioms only apply when the actual argument expressions and the result identifier are the same as the formals used in the specifications; hence one must also use the previous rule, in general.
The axiom used for a message-passing expression during verification is determined by the nominal types of the argument expressions (that is, using static instead of dynamic overloading).

Because of the above simplifications, the following inference rule does the real work for a language without subtyping.

\[
\begin{align*}
\vdash R_1 & \land \cdots \land R_n \ \{ y \gets E \} \ Q[z/x] \\
\vdash P \ \{ v_1 \gets E_1 \} \ (R_1[v_1/x_1])[x/z], \\
\vdots \\
\vdash P \ \{ v_n \gets E_n \} \ (R_n[v_n/x_n])[x/z] \\
\vdash P \ \{ y \gets (\text{fun} \ (x: S) \ E_0) \ (E_1, \ldots, E_n) \} \ Q
\end{align*}
\]

The rule as a whole says that to prove that the desired triple holds, one chooses conjuncts \( R_i \) that are sufficient to prove the desired post-condition from the body of the function abstract. Then one shows that these conjuncts characterize the argument values. For example, to prove formula (3), it suffices to prove the following triples, where \( i_1 \) and \( i_2 \) have nominal type \( \textbf{Integer} \).

\[
\begin{align*}
(f1 == 3) & \land (f2 == 4) \\
\{ f \gets \text{add}(f1, f2) \} \\
 f & == 7 \\
\{ i1 == 3 \} \ & i1 == 3 \\
\{ i2 == 4 \} \ & i2 == 4
\end{align*}
\]

The assertions \( R_i \) may contain the formal argument identifiers, \( x_i \), and thus may be written using the trait functions defined on the types \( S_i \). The assertions \( R_i[v_i/x_i] \) will type-check because the nominal type of \( v_i \) is the nominal type of \( E_i \), which must be a subtype of \( S_i \) (i.e., the type of \( x_i \)). It is crucial for the soundness of this rule that whenever \( R_i[v_i/x_i] \) holds, then \( R_i \) holds as well. (The requirements placed on trait functions for subtypes in Section 6 ensure that this condition is met.) The idea is that \( R_i[v_i/x_i] \) characterizes the argument \( E_i \) at its nominal type (the type of \( v_i \)), while the type of \( x_i \) is a supertype of \( E_i \)’s type.

An unusual feature of our formal system is that the rule of consequence

\[
\begin{align*}
\vdash (P \Rightarrow P_1), \vdash P_1 \ \{ y \gets E \} \ Q_1, \vdash (Q_1 \Rightarrow Q) \\
\vdash P \ \{ y \gets E \} \ Q
\end{align*}
\]

is only valid when the assertions involved are subtype-constraining. This restriction is necessary, as can be seen by the following example. Consider the implication

\[
\text{numerator}(f) = 0 \Rightarrow (f = 0/1),
\]

where \( f \) has nominal type \( \textbf{Fraction} \). This implication can be proved from the axioms of the trait \( \textbf{IntAndRat} \), which means that if \( f \) denotes a \( \textbf{Fraction} \), then the implication is valid. However, it is not valid if \( f \) denotes the \( \textbf{Integer} \) with value \( 0 \). A solution is to use \( "==" \) instead of the second \( "==" \) to obtain a subtype-constraining assertion.

The other rules of the logic are fairly straightforward or standard.

Our verification method allows a function implementation to be verified once, without considering the different combinations of actual argument types. Instead, a function implementation is verified as if the actuals had the types specified for the formals. For example, the correctness of an implementation of \( \text{sqrt} \) would be verified by reasoning about the formal argument \( x \) as if it were a fraction. Such a verification guarantees correctness for arguments of a subtype, because of the semantic restrictions on subtype relations. (Termination of recursive functions must be verified separately.)

### 6 Soundness of the Method

The soundness of the verification method discussed above rests on the syntactic restrictions on \( \texttt{ResType} \) and \( \leq \), the semantic restrictions on \( \leq \) and \( \mathcal{R} \) and the following technical results [Lea90]:

- Each expression of nominal type \( \text{T} \) can only denote objects of a type \( S \leq \text{T} \). This is ensured by type checking and the syntactic constraints on type specifications.

- An assertion \( P \) characterizing the values of actual parameters \( v_i \) holds for the corresponding formals \( x_i \), provided \( P \) with the \( x_i \) substituted for the \( v_i \) type checks. This is ensured by dynamic overloading of trait functions. For example, suppose “numerator(j) = 3” describes an actual parameter \( j \), to which the formal \( f \) is bound; then “numerator(f) = 3” also holds.

- Subtype-constraining assertions that can be proved from the traits used in a type specification remain valid when an identifier \( x \) is allowed to refer to the values of a subtype of the nominal type of \( x \). This property is ensured by semantic constraints on \( \mathcal{R} \). For example, the implication

\[
\text{numerator}(f) = 0 \Rightarrow (f == 0/1),
\]
operations, but also for assertions and programs. In the study of the lambda calculus, this kind of theorem is known as the fundamental theorem of logical relations [Sta85] [Mit86]. Showing that the substitution property holds for assertions is crucial to proving the soundness of the verification system.

Disciplined use of subtypes cannot lead to surprising program behavior, because the substitution property also holds for program expressions and recursively defined program functions. Indeed, the relationships are preserved\(^3\) even if functions and operations are permitted to be nondeterministic [Lea90].

7 Related Work

Ours is the first formal verification technique for object-oriented programs that use message-passing that has been proven to be sound [Lea89] [Lea90].

Cardelli was the first to formally describe subtype relationships and type checking for a fixed set of types [Car84]. Our work generalizes Cardelli’s to abstract data types. That is, given appropriate specifications of the types Cardelli discusses, the subtype relationships Cardelli describes for immutable record and variant types are also subtype relationships in our sense [Lea90]. However, our notion of subtypes is based on type specifications, and thus can handle arbitrary immutable abstract types.

Bruce and Wegner [BW87] use coercion functions with a substitution property, which are like our simulation relations, to give a definition of subtype relations. However, they do not discuss reasoning about object-oriented programs. Using relations instead of functions allows us to handle an abstract type whose space of abstract values is not reduced (in the sense that objects with two distinct abstract values may behave the same). Examples can be found in [Lea90]. Bruce and Wegner also do not handle operations that may fail to terminate.

For the language Eiffel [Mey88], Meyer requires that the pre-condition of an instance operations of a super class \( T \) must imply the pre-condition of the instance operation of the instance operation of each subclass of \( T \) with the same name; furthermore, the post-condition of the subclass’s operation must imply the post-condition of \( T \)’s operation. However, assertions for Eiffel specifications are written using a type’s operations. A subclass in Eiffel can redefine the operations of a superclass, so that while the implications among the pre- and post-conditions may be valid, the

---

\(^3\) The relationships of \( R \) are preserved by a nondeterministic operation if for each possible result on the left hand side of Formula (16), there is some possible result on the right hand side for which the required relationship holds.
behavior of instances of the subtype may be surprising. The extreme of this problem occurs for deferred types: types for which one or more of the operations are not implemented (i.e., their implementation is deferred to a subclass). Consider a class \( D \) where all the operations are deferred. The pre- and post-conditions of the operations of \( D \) are written using the operations of \( D \). But the operations of \( D \) are not implemented, so the assertions that are used to define these operations are meaningless. We can specify such deferred types, because the trait functions used to specify operations are specified independently of the operations.

P. America has independently developed a definition of subtype relationships [Ame89]. Types are specified by describing the abstract values of their instances, and the post-condition of each operation relates the abstract values of the arguments to the abstract value of the result. The “trait functions” used to describe a supertype’s abstract values need not be defined for the subtype’s abstract values. 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

\[
\text{Pre}(	ext{Super}) \circ f \Rightarrow \text{Pre}(	ext{Sub}) \quad (17)
\]

\[
\text{Post}(	ext{Sub}) \Rightarrow \text{Post}(	ext{Super}) \circ f \quad (18)
\]

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. As with Reynolds and Bruce and Wegner, since \( f \) must be a function, the set of abstract values must be reduced, otherwise there might not be a transfer function.

America’s definition of subtyping handles mutable types, but not aliasing. America’s type specifications do not have class operations, they only have instance operations. The lack of class operations makes it difficult to specify types whose objects are created in one of several states. Because of the lack of class operations, America’s notion of subtype is identical to the notion of refinement. A type \( S \) is a refinement of \( T \) if each implementation of \( S \) is an implementation of \( T \). We allow class operations but do not require that a subtype implement the class operations of its supertypes. So for us, a type can be a subtype without being a refinement, although a refinement is necessarily a subtype.

8 Future Work

One area of future work is extending our approach to deal with mutable types. Also needed for practical use are symbolic methods for proving subtype relationships. Another area is the verification of implementations of classes that use inheritance. Finally, programs that test the types of objects are currently beyond the capabilities of our verification method. The problem is that functions that test argument types violate data abstraction and can thus behave differently on different types of arguments.

9 Conclusions

We have described, and illustrated with a simple example, a method for specifying and verifying object-oriented programs that use subtypes and message-passing. This method applies directly to applicative languages with immutable data types, but can be easily extended to handle assignments.

Since subtyping imposes strong conditions on the behavior of the types involved, it seems necessary to design subtypes with subtyping in mind. Such strong conditions also seem necessary for the soundness of modular program verification, so that one can reason about subtypes implicitly. Hence, we suggest that subtype relationships should be declared, rather than inferred on the basis of structural information such as signatures [BHHJ86] or subclass (inheritance) relationships among implementations [BDMN73].

Reasoning based on subtyping and nominal type information seems to be used informally by programmers working with object-oriented languages [Sny86]. However, it is important for programmers to recognize that subtyping is a rather strong behavioral constraint that is independent of sublassing.

The principal advantage of our approach is that it allows modular reasoning. Functions are specified only once, and the form of a function specification is independent of subtype relationships. In addition, the verification of a function implementation proceeds as if the actual arguments’ types are the same as the types of the formal arguments. Therefore, new subtypes may be added to a program without affecting function specifications or the correctness of their implementations.

10 Acknowledgements

Thanks to Barbara Liskov for urging us to describe simulation as a relationship among abstract values that is specified along with types. Thanks to John Guttag for suggesting the use of dynamic overloading.
for specifications. Thanks to Kelvin Nilsen, William Cook, T. B. Dinesh, Carl Waldspurger, and the referees for comments on various drafts.

References


