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

*The work of both authors was supported in part by the National Science Foundation under Grant CCR-8716884, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988. While a graduate student at MIT, Leavens was also supported in part by a GenRad/AEA Faculty Development Fellowship, and at ISU he has been partially supported by the ISU Achievement Foundation.

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 [Lea90]. 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 **fun** instead of 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 *slanted* font to distinguish them from message names written in **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 [Rey80] [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 *ResType*, which maps message names and tuples of types to expected result types, and a user-specified reflexive and transitive relation among types, $\leq$, called the *subtype relation*.

```
⟨program⟩ ::= prog ( ⟨decls⟩ ) : ⟨type⟩ =
              ⟨expr⟩
            | ⟨rec fun def⟩ ⟨program⟩

⟨decls⟩ ::= ⟨decl list⟩ | ⟨empty⟩
⟨decl list⟩ ::= ⟨decl⟩ | ⟨decl list⟩ , ⟨decl⟩

⟨decl⟩ ::= ⟨identifier⟩ : ⟨type⟩

⟨empty⟩ ::=

⟨expr⟩ ::= ⟨identifier⟩
         | ⟨message name⟩ ( ⟨expr list⟩ )
         | ⟨function identifier⟩ ( ⟨exprs⟩ )
         | ( ⟨function abstract⟩ ) ( ⟨exprs⟩ )
         | if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩ fi
         | ( ⟨expr⟩ )

⟨exprs⟩ ::= ⟨expr list⟩ | ⟨empty⟩
⟨expr list⟩ ::= ⟨exprs⟩ | ⟨expr list⟩ , ⟨expr⟩

⟨function abstract⟩ ::= fun ( ⟨decls⟩ ) ⟨expr⟩

⟨rec fun def⟩ ::= fun ⟨function identifier⟩
      ( ⟨decl list⟩ ) : ⟨type⟩ = ⟨expr⟩ ;
```

Figure 1: Programming language syntax.

For example, consider the message-passing expression `add(a,b)`. The nominal types of the arguments `a` and `b` are given in their declarations, say `Fraction` and `Integer`. The nominal type of the result is then *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, *ResType* must be monotone in the following sense: for all message names $\mathbf{g}$, and for all tuples of types $\vec{S} \leq \vec{T}$, if *ResType*$(\mathbf{g}, \vec{T})$ is defined, then so is *ResType*$(\mathbf{g}, \vec{S})$, and *ResType*$(\mathbf{g}, \vec{S}) \leq$ *ResType*$(\mathbf{g}, \vec{T})$. (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. Wadler and Blott's method dictionaries provides 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 [ASS85, 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.

```
prog (b: Boolean):  Integer =
    num(sqrt(if b
             then 16
             else create(Fraction,3,4)
             fi))
```

Figure 3: Call to *sqrt* that shows the general case of message-passing.

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 **lt**, **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 **lt**, **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 $0 \leq x$
ensures $(0 \leq r)$ & $(|(r * r) - x| \leq (1/100))$

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 formals. 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 "$\leq$", "$|\cdot|$", "*", "_", 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 "$(0 \leq r)$ & $(|(r * r) - 16| \leq (1/100))$". Hence the value of the result `r` 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 [Lea89].

```
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,R → R
            #==#: R,R → Bool
            #==#: Int,Int → Bool
            #==#, #≤#, #≥#, #>#, #<#
                : R,Int → Bool
            #==#, #≤#, #≥#, #>#, #<#
                : Int,R → Bool
    constrains |#|, gcd, numerator, denominator,
            #==#, #*#: R,R → 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*d, m*d) = 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
```

Figure 5: The trait IntAndRat.

```
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)
        ensures 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(c:IntegerClass) 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)
```

Figure 7: Specification of the type **Integer**.

$\leq$ among type symbols (the subtype relation), and a family of relations $\mathcal{R}$ 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 **Integer** states that **Integer** is a subtype of **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 **subtype of** relationships given in the type specifications.

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

The family $\mathcal{R}$ 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 $\mathcal{R}$. That is, if $\mathtt{S} \leq \mathtt{T}$, then it is required that for every object of type S, its abstract value is related by $\mathcal{R}_{\mathtt{T}}$ to the abstract value of some object of type T, and that $\mathcal{R}$ 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 **Integer** have **Integer** as the type of their second argument, and most have **Integer** as the type of their result. Thus, for example, if **a** and **b** denote objects of type **Integer**, then **add(a,b)** must denote an **Integer**. The operation specification that determines the behavior of an invocation of **add** is the most specific specification whose argument types are supertypes of the types of the actual arguments, because message-passing is used at run-time. For example, the result of **add(a,b)** need only satisfy the specification of the **add** operation of **Integer** if **a** and **b** denote **Integers**. On the other hand, if **a** denotes a **Fraction**, then the result of **add(a,b)** is determined by the specification of the **add** operation of **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 = (f1 / 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 $\leq$ 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 applicative, we use a Hoare logic for program verification, because we are ultimately interested in verification of imperative programs.

Hoare-triples are written $P\ \{\mathtt{v} \leftarrow E\}\ Q$ 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, $P\ \{\mathtt{v} \leftarrow E\}\ Q$ 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.

$$\frac{\vdash P\ \{\mathtt{y} \leftarrow (\mathtt{fun}\ (\vec{\mathtt{x}} : \vec{S})\ g(\vec{\mathtt{x}}))\ (\vec{E})\}\ Q}{\vdash P\ \{\mathtt{y} \leftarrow g(\vec{E})\}\ Q} \quad (1)$$

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

$$\text{true}\ \{\mathtt{f} \leftarrow \mathtt{add(3,4)}\}\ \mathtt{f} == 7 \quad (2)$$

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

$$\begin{array}{l}\text{true}\\ \{\mathtt{f} \leftarrow (\mathtt{fun}\ (\mathtt{f1},\mathtt{f2})\ \mathtt{add(f1,f2)})\ (3,4)\}\\ \mathtt{f} == 7\end{array} \quad (3)$$

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**:

$$\vdash\ \text{true}\ \{\mathtt{f} \leftarrow \mathtt{add(f1,f2)}\}\ \mathtt{f} == (\mathtt{f1} + \mathtt{f2}) \quad (4)$$

$$\vdash\ \text{true}\ \{\mathtt{i} \leftarrow \mathtt{add(i1,i2)}\}\ \mathtt{i} = (\mathtt{i1} + \mathtt{i2}). \quad (5)$$

The specification of *sqrt* generates the following axiom:

$$\vdash\ \begin{array}{l}0 \leq \mathtt{x}\\ \{\mathtt{r} \leftarrow sqrt(\mathtt{x})\}\\ (0 \leq \mathtt{r}) \& (|(\mathtt{r} * \mathtt{r}) - \mathtt{x}| \leq (1/100))\end{array} \quad (6)$$

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[2] does the real work for applications. This rule would be different in a language without subtyping.

$$\frac{\begin{array}{c}\vdash R_1 \,\&\, \cdots \,\&\, R_n\ \{\mathtt{y} \leftarrow E_0\}\ Q[\vec{\mathtt{z}}/\vec{\mathtt{x}}] \\ \vdash P\ \{\mathtt{v}_1 \leftarrow E_1\}\ (R_1[\mathtt{v}_1/\mathtt{x}_1])[\vec{\mathtt{x}}/\vec{\mathtt{z}}], \\ \vdots \\ \vdash P\ \{\mathtt{v}_n \leftarrow E_n\}\ (R_n[\mathtt{v}_n/\mathtt{x}_n])[\vec{\mathtt{x}}/\vec{\mathtt{z}}]\end{array}}{\vdash P\ \{\mathtt{y} \leftarrow (\mathtt{fun}\ (\vec{\mathtt{x}} : \vec{\mathtt{S}})\ E_0)\ (E_1, \ldots, E_n)\}\ Q} \tag{7}$$

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 `i1` and `i2` have nominal type `Integer`.

$$\begin{array}{l}(\mathtt{f1} == 3)\ \&\ (\mathtt{f2} == 4) \\ \{\mathtt{f} \leftarrow \mathtt{add(f1, f2)}\} \\ \mathtt{f} == 7\end{array} \tag{8}$$

$$\text{true}\ \{\mathtt{i1} \leftarrow \mathtt{3}\}\ \mathtt{i1} == 3 \tag{9}$$

$$\text{true}\ \{\mathtt{i2} \leftarrow \mathtt{4}\}\ \mathtt{i2} == 4 \tag{10}$$

The assertions R_i may contain the formal argument identifiers, $\mathtt{x}_i$, and thus may be written using the trait functions defined on the types $\mathtt{S}_i$. The assertions $R_i[\mathtt{v}_i/\mathtt{x}_i]$ will type-check because the nominal type of $\mathtt{v}_i$ is the nominal type of E_i, which must be a subtype of $\mathtt{S}_i$ (i.e., the type of $\mathtt{x}_i$). It is crucial for the soundness of this rule that whenever $R_i[\mathtt{v}_i/\mathtt{x}_i]$ 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 $R_i[\mathtt{v}_i/\mathtt{x}_i]$ characterizes the argument E_i at its nominal type (the type of $\mathtt{v}_i$), while the type of $\mathtt{x}_i$ is a supertype of E_i's type.

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

$$\frac{\vdash (P \Rightarrow P_1), \vdash P_1\ \{\mathtt{y} \leftarrow E\}\ Q_1, \vdash (Q_1 \Rightarrow Q)}{\vdash P\ \{\mathtt{y} \leftarrow E\}\ Q}$$

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

$$(\text{numerator(f)} = 0) \Rightarrow (\text{f} = 0/1), \tag{11}$$

where "f" has nominal type `Fraction`. This implication can be proved from the axioms of the trait `IntAndRat`, which means that if "f" denotes a `Fraction`, then the implication is valid. However, it is not valid if "f" denotes the `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 *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 *ResType* and $\leq$, the semantic restrictions on $\leq$ and $\mathcal{R}$ and the following technical results [Lea90]:

- Each expression of nominal type `T` can only denote objects of a type `S` $\leq$ `T`. This is ensured by type checking and the syntactic constraints on type specifications.
- An assertion P characterizing the values of actual parameters $\mathtt{v}_i$ holds for the corresponding formals $\mathtt{x}_i$, provided P with the $\mathtt{x}_i$ substituted for the $\mathtt{v}_i$ 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

$$(\text{numerator(f)} = 0) \Rightarrow (\text{f} == 0/1), \tag{12}$$

[2] The notation $(R_i[\mathtt{v}_i/\mathtt{x}_i])[\vec{\mathtt{x}}/\vec{\mathtt{z}}]$ means the formula R_i with $\mathtt{v}_i$ replacing $\mathtt{x}_i$ throughout and then each $\mathtt{x}_i$ substituted for $\mathtt{z}_i$. Fresh identifiers $\vec{\mathtt{z}}$ are used to hide bindings of $\vec{\mathtt{x}}$ in the assertions that characterize the arguments to the function abstract, so that in reasoning about E_0 the proper scope applies. That is, bindings of the $\mathtt{x}_i$ in P or Q do not mean the same $\mathtt{x}_i$ that are local to the function abstract. The identifiers $\mathtt{z}_i$ must be fresh and the result identifier `y` must not be one of the $\mathtt{x}_i$; these restrictions avoid capture problems.

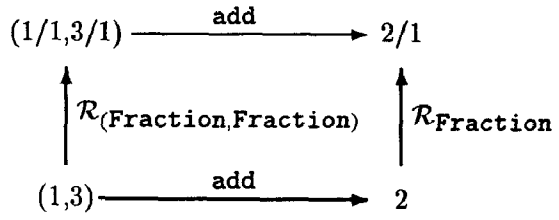

Figure 8: The substitution property for add.

is valid even if the value of **f** is an **Integer**.

- If $q\ \mathcal{R}_{\mathtt{T}}\ r$, then a subtype-constraining assertion P characterizing the value of **x** : T holds when **x** is bound to q if and only if P holds when **x** is bound to r. This property is ensured by semantic constraints on $\mathcal{R}$.

The most important constraint on the family $\mathcal{R}$ is the substitution property — that simulation relationships are preserved by both message-passing and the trait functions. For example, the family $\mathcal{R}$ described in the specifications of **Integer** and **Fraction** has the substitution property, since the following relationships, among others, hold:

$$\texttt{add}(1,3)\ \mathcal{R}_{\texttt{Fraction}}\ \texttt{add}(1/1,3/1) \quad (13)$$

$$\texttt{sub}(1,3)\ \mathcal{R}_{\texttt{Fraction}}\ \texttt{sub}(1/1,3/1) \quad (14)$$

$$\text{numerator}(3)\ \mathcal{R}_{\texttt{Integer}}\ \text{numerator}(3/1) \quad (15)$$

The first relationship is illustrated by the commutative diagram in Figure 8 (assuming that add for **Fractions** returns a **Fraction**). These relationships can be verified using the specifications of **Integer** and **Fraction**, regardless of the implementations of those types.

More formally, $\mathcal{R}$ has the substitution property if and only if the following holds: let T be a type, and let $\vec{S}$, $\vec{U}$, and $\vec{V}$ be tuples of types such that $\vec{U} \leq \vec{S}$ and $\vec{V} \leq \vec{S}$; then for all tuples of values $\vec{q} : \vec{U}$ and $\vec{r} : \vec{V}$ such that $\vec{q}\,\mathcal{R}_{\vec{S}}\,\vec{r}$, and for all trait function symbols or message names g such that $ResType(g, \vec{S}) = \mathtt{T}$,

$$g(\vec{q})\ \mathcal{R}_{\mathtt{T}}\ g(\vec{r}). \quad (16)$$

A family $\mathcal{R}$ constructed as described above is a *simulation relation* if it satisfies the substitution property. The construction of $\mathcal{R}$ ensures other desirable properties. Bruce and Wegner have stated a similar list of properties for their coercer functions [BW87], as does Reynolds [Rey80].

The semantics constraints on $\leq$ require that the specified family $\mathcal{R}$ is a simulation relation.

If $\mathcal{R}$ is a simulation relation, then the substitution property holds not just for single trait functions and 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 superclass 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 $\mathcal{R}$ 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

$$\text{Pre(Super)} \circ f \Rightarrow \text{Pre(Sub)} \quad (17)$$

$$\text{Post(Sub)} \Rightarrow \text{Post(Super)} \circ f \quad (18)$$

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 [BHJL86] 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 subclassing.

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

[Ame89] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. Technical Report 443, Philips Research Laboratories, Nederlandse Philips Bedrijven B. V., January 1989.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Mass., 1985.

[BDMN73] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *SIMULA Begin.* Auerbach Publishers, Philadelphia, Penn., 1973.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. *ACM SIGPLAN Notices*, 21(11):78–86, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[BJ82] Dines Bjorner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice-Hall International, London, 1982.

[BW87] Kim B. Bruce and Peter Wegner. An algebraic model of subtype and inheritance. To appear in *Database Programming Languages*, Francois Bancilhon and Peter Buneman (editors), Addison-Wesley, Reading, Mass., August 1987.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In D. B. MacQueen G. Kahn and G. Plotkin, editors, *Semantics of Data Types: International Symposium, Sophia-Antipolis, France*, volume 173 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, New York, N.Y., June 1984. A revised version of this paper appears in *Information and Computation*, volume 76, numbers 2/3, pages 138–164, February/March 1988.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[GH86a] J. V. Guttag and J. J. Horning. A Larch shared language handbook. *Science of Computer Programming*, 6:135–157, 1986.

[GH86b] J. V. Guttag and J. J. Horning. Report on the Larch shared language. *Science of Computer Programming*, 6:103–134, 1986.

[GM87] Joseph A. Goguen and Jose Meseguer. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. Technical Report CSLI-87-92, Center for the Study of Language and Information, March 1987.

[Gog84] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley Publishing Co., Reading, Mass., 1983.

[Gut80] John Guttag. Notes on type abstractions (version 2). *IEEE Transactions on Software Engineering*, SE-6(1):13–23, January 1980. Version 1 in *Proceedings Specifications of Reliable Software*, Cambridge, Mass., IEEE, April, 1979.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison Wesley, Reading, Mass., 1989.

[Lea89] Gary Todd Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989. The author's Ph.D. thesis.

[Lea90] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Department of Computer Science, Iowa State University, July 1990.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., 1988.

[Mit86] John C. Mitchell. Representation independence and data abstraction (preliminary version). In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 263–276. ACM, January 1986.

[Rey80] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, January 1980.

[Rey85] John C. Reynolds. Three approaches to type structure. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin. Volume 1: Colloquium on Trees in Algebra and Programming (CAAP '85)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, New York, N.Y., March 1985.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[Sta85] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2/3):85–97, May/June 1985.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[Win83] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Win87] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.