INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Using Value Semantic Abstractions to Guide Strongly Typed Library Design

B. Gomes, D. Stoutamire, B. Weissman and J. Feldman

TR-97-061

December 1997

Abstract

This report addresses typing problems that arise when modelling simple mathematical entities in strongly typed languages such as Sather, which are eliminated by a proper distinction between mutable and immutable abstractions. We discuss the reasons why our intuition leads us astray, and provide a solution using statically type-safe specialization through constrained overloading. We also discuss the type relationships between mutable and immutable classes and the notion of freezing objects.

1.0 Introduction

When modelling mathematical entities - such as triangles and polygons, sets and bags, integers and complex numbers - in object oriented languages, confusion often arises as to the nature of the typing relationships between these entities. Inspite of the clean is-a relationships between these entities they do not appear to be substitutable. In strongly (statically) typed languages, violations of substitutabilty often manifest themselves as problems in type-conformance. These violations have, in part, been responsible for the ever-present co- vs. contravariance debate.

Behind the co-vs.contravariance debate, and object-oriented programming as a whole, is the notion that humans think naturally in terms of objects and, therefore, that the use of this metaphor is an aid in modelling systems when programming. However, it superficially appears that when we map mathematical objects, which we understand quite well, using the object oriented metaphor, many of our intuitive categorizations and inferences from the world of mathematics break down.

This report presents a non-theoretical description of the problem of this error in intuition, its consequences, and solutions. The theory behind these relationships is described in [Cas95] but the degree of formalism obscures what is basically a simple, but important, point. Our own views arose independantly from these theoretical considerations, during the very practical exercise of designing the Sather libraries in a type-safe manner.

In short, this report provides a detailed answer to question 21.8 from the C++ FAQ by Marshall Cline, from “www.cis.ohio-state.edu/hypertext/faq/usenet/C++-faq”:

> But I have a Ph.D. in Mathematics, and I’m sure a Circle is a kind of an Ellipse! Does this mean Marshall Cline is stupid? Or that C++ is stupid? Or that OO is stupid?

Similar comments may be found in [Mar96] regarding squares and rectangles and in [Mey96] regarding rectangles and polygons.

Section 3.0 describes how the problem arises from a basic error in modelling mathematical objects in standard object oriented languages. Avoiding the error involves the use of abstract data types with immutable semantics, along with the judicious use of specialization. Subsequent sections explore the relationship between data types with mutable and immutable semantics, and the nature of the overloading rules needed to support specialization. Along the way, the dangers of poor naming and the importance of right subtyping are addressed.

Acknowledgements

Many thanks to Welf Lowe and Wolf Zimmerman for helpful comments on this report.

1.1 Conformance and Substitutability

The notion of substitutability is central to our discussion. By substitutability we mean that objects of type A may be safely replaced by objects of any subtype of A [Lis88], [Mar96] . Complete substitutability only occurs if the semantics of the subtype are substitutable for the semantics of the

supertype. A portion of the method semantics may be expressed through the method signatures and type checked. However, in order to understand whether a method is truly substitutable for the corresponding method in a supertype, we sometimes need to consider more details of the method semantics. Additonal semantic information may be expressed through method pre- and post- conditions.

Note that we are not concerned so much with the actual expression of pre- and post-conditions in the language or libraries; rather, we use them as an aide in understanding the semantics of a method and/or a class, and thus in addressing the issue of substitutability.

Pre conditions

The preconditions of a method are the set of conditions that must be met to permit the method to execute correctly. The preconditions essentially capture what the method expects to be true when it is called - a violation of the precondition denotes a bug in the client of the method [Mey94].

In order to ensure substitutablity of a subtype for its supertype, the methods in the subtype must execute correctly whenever the preconditions for the supertype method are met. Thus, the preconditions of the subtype must be implied by the preconditions of the supertype. Substitutability requires that when a class SUB_FOO with method SUB_FOO::bar subtypes from a class SUPER_FOO:

> preconditions(SUPER_FOO::bar) must imply preconditions(SUB_FOO::bar)

More intuitively, to ensure the substitutability of a subclass, **preconditions may only be weakened** under subtyping**.**

Note that invariant preconditions are sufficient for most purposes; it is very rare for a subtype to actually weaken its preconditions. However, we retain the notion of weakening preconditions in order to remind us of the direction of the implication.

Post conditions

Postconditions are the converse of preconditions - they are a statement of what the method guarantees to the caller after it is done. An error in a postcondition denotes an error in the method implementation. In order to guarantee substitutability of a subtype for its supertype, the postconditions of the subtype method must imply the postconditions of the supertype method.

> postconditions(SUB_FOO::bar) must imply postconditions(SUPER_FOO::bar)

More intuitively, **postconditions may only be strengthened** under subtyping.

We use an Eiffel/Sather based syntax to state pre and post conditions:

```
class REAL is
   sqrt:REAL;
      precondition self  >= 0;
      postcondition  result*result = initial(self);
```

The postcondition may make use of the the special variable result whose value is set to the return value of the method. It is sometimes also necessary to compare values from before and after the method execution. Since the postcondition is evaluated after the method terminates, we use the special form initial(<expression>); to obtain the value of <expression> from before the method execution. In the above example, the postcondition states that the return value of the square root function when multiplied by itself must be equal to the initial value of self.

Pre and post conditions may also be thought of as assertions that are directly visible in the method interface.

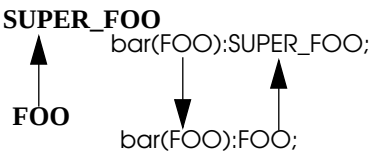
Class invariants

A class invariant expresses constraints that must always hold true of any object of that class (except, possibly, during the course of a method invocation on the class). Class invariants behave like postconditions - the child's invariant must imply the parent's invariant. In the subtyping diagrams used througout this report, the direction of the arrows between sub- and super-type reflects the direction of this class invariant implication.

Type-safe subtyping

The types of the formal arguments to a method are a special case of preconditions[1] - to ensure substitutability of a subtype, the declared arguments of a subtype method must be implied by (i.e. be supertypes of, or *contravariant* with) the declared arguments of the corresponding subtype method. Likewise, the return type of a method is a special case of a postcondition. To ensure substitutablity of a subtype, the return type of the subtype method must imply (i.e. be a subtype of, or *covariant* with) the return type of the corresponding supertype method. Intuitively, the preconditions (argument types) may become less restrictive and the postcondition (return type) may become more restrictive. In the example below, the signature SUB_FOO::bar must have an argument that is of type FOO or SUPER_FOO. The return type must be either FOO or SUB_FOO.

```
class SUPER_FOO
   bar(FOO):SUPER_FOO;

class FOO
    subtypes from SUPER_FOO
   bar(FOO):FOO;
```



This report deals with strongly typed languages, i.e. languages in which the subtyping rule verifies the type-safety of the substitutability statically[2] [Lis88]. The same issues are relevant to object oriented design in weakly-typed languages, but do not manifest themselves as readily in the type-system.

1. The formal type of a method argument states the precondition that the type of the actual argument must be a subtype of the formal type for correct method execution.
2. No run-time typing errors can occur in a language such as Sather, except in a typecase.

1.2 Syntax Usage

In the discussion below, while our concerns were driven by Sather, the resulting issues are by no means limited to Sather. Hence, we have used a fairly generic pseudo-code that is somewhat more explicit than direct Sather syntax that should be more understandable to users of other similar, languages.

As a note for users of C++ - self as used here is equivalent to this in C++ and the term abstract class is equivalent to a virtual class with all virtual methods. Sather additionally completely separates the concepts of subtyping and code inclusion (which other practictioners sometimes refer to as inheritance). To avoid confusion, we do not use the term inheritance, and since this discussion focuses on typing issues, we omit any discussion of code inclusion.

Other Languages

Of the other languages in the same general space as Sather (efficient, type-safe, object oriented languages), C++ is strongly typed, but has the more restrictive typing rule of no-variance. Java is similarly restricted. Eiffel supports the covariant rule which does not permit compile-time type safety, though there have been recent moves in this direction [Mey96].

2.0 The Problem

The power of object-oriented programming arises, in part, because it permits the expression of existing object classifications through subtyping relationships. The tools and intuitions regarding these existing classifications may then be used when reasoning about the program, resulting in code that is easier to understand, maintain and extend. However, in many seemingly straight-forward mathematical contexts, reflecting standard mathematical is-a relationships in the subtyping hierarchy leads to problems with substitutability.

The problem is essentially the presence of specialization in the natural inheritance hierarchy. A specialization from a class A to a class B is a relationship in which the class invariant in B more restrictive than than the invariant of B. If B is a specialization of A, then $Inv_B => Inv_A$. The problem with such specialization is that method calls that are legal in A may be problematic if B is substituted for A, since they may result in a violation of the more restrictive class invariant. Thus, methods in B have more restrictive preconditions i.e. $Pre_{m,B} => Pre_{m,A}$. Suppose B extends the class invariant of A with the predicate *P*, then B has the stronger invariant $inv_A \wedge P$. A problem will arise with substitutability if the postcondition of a method m in A contradicts *P*, then the postcondition of m also contradicts the invariant of B and m cannot exist in B. If m cannot exist in B, then B cannot be a substitutable for A.

This problem may be illustrated using polygons and triangles, an illustration found in many introductory texts on object oriented programming

```
class POLYGON
   n_points:INT;
   add_point(point:POINT);
      postcondition  n_points = initial(n_points)+1;
```

In the above definition, the point addition method modifies the polygon, resulting in a polygon with one more point. A triangle may then be considered a particular kind of polygon

```
class TRIANGLE subtypes from POLYGON
   n_points:INT;
   add_point(point:POINT);
```

If we consider the invariant of a polygon to be n_points > 2, then a triangle is a polygon with the class invariant that the n_points = 3. In the case of the triangle, it is not at all clear what the add_point method should do. In other words, in the postcondition of add_point, n_points may be greater than 3, which contradicts the invariant of B, namely n_points = 3. Hence, add_point cannot exist in the triangle and therefore triangle cannot be a subtype of polygon.

- Raise an exception. The problem with this solution is that the POLYGON::add_point method does not raise an exception. Hence, if we were to substitute a triangle for a polygon object, unexpected exceptions might occur. Raising exceptions that could not be raised in the superclass may be viewed as a special case of violating the postcondition of the method.
- Permit the user to “undefine” the add_point method of the triangle. Once again, substitutability is violated and run-time type errors may result if the method add_point is called on a variable of type POLYGON.
- Eliminate the offending method from POLYGON. The supertype no longer has a method whose post-condition violates the predicate added to the invariant in the subtype. This works and is a correct solution, but it still does not explain why the mathematical subtyping relationship cannot be expressed.
- Eliminate the subtyping relationship. This works too, but, as question 28.1 of the C++ faq asks, why can’t we subtype when mathematically a triangle really is a kind of polygon.

None of these solution is pleasant; a clean mathematical relationship cannot be cleanly modelled in the type-system. The problem with circles and ellipses is identical to the triangle/polygon case mentioned above.

3.0 The Real Problem

The real problem lies in a difference between the mathematical conception of objects, and the standard object-oriented conception. This is the familiar distinction between values and references in a slightly different disguise. The mathematical notion of a polygon is fundamentally immutable. A new polygon may arise by considering a particular triangle and an addi-

tional point. However, this does not ever modify the original triangle. The issue is obvious when you consider more basic mathematical entities such as the number three - adding and subtracting values never modify the number three.

The problem is partially one of education - object oriented practitioners are used to thinking in terms of persistant, modifiable objects, which is quite different from the platonic objects in the world of mathematics. The point we wish to stress is that there is no problem with mathematical hierarchies as we think of them, nor is there any problem with object oriented programming. There is, however, a problem, with modelling mathematical entities as if they were modifiable and expecting the mathematical hierarchies to continue to hold.

In the context of rectangles and polygons and also ostriches and birds, [Mey94] says:

> I should note in passing that some people criticize [method overriding] as incompatible with a good use of inheritance. They are deeply wrong. It is a sign of the limitations of the human ability to comprehend the world -- similar perhaps to undecidability results in mathematics and uncertainty results in modern physics -- that we cannot come up with operationally useful classifications without keeping room for some exceptions. Descendant hiding is the crucial tool providing such flexibility. Hiding add_vertex from RECTANGLE or fly from OSTRICH is not a sign of sloppy design; it is the recognition that other inheritance hierarchies that would not require descendant hiding would inevitably be more complex and less useful.

There is something disturbing about this notion; the efficacy of object-oriented programming depends in part, at least, on the belief that humans think in terms of objects and that the intuition from human objects can drive an object-oriented type hierarchy. If this is not so, if our human intuitions are fraught with errors, this spells trouble for the metaphorical basis behind object-oriented programming.

We would like to note that the two problems - that of an ostrich being a subtype of birds and rectangle being a subtype of polygon are quite different. If we are basing the bird hierarchy on the common notion of birds, then the common human notion also notes that there are exceptional birds such as penguins, emus and ostriches that do not fly. The point is that the exception is noted in the guiding human hierarchy as well. If the bird hierarchy is based on a more precise biological notion of birds, then flying will not be a property of birds to begin with, and the problem does not arise.

However, we never think of the relationship between rectangles and polygons as being in any way exceptional. Exceptions only arise when we try to model unchanging mathematical entities using modifiable objects - all bets are off, and implications from the world of mathematical polygons may well be violated in this brave new world of modifiable polygons. This is not to say that mutable polygons are useless or “wrong” - just that they are a different concept from the mathematical entities we are used to; modifiable polygons should not be called polygons (mathematics has precedence, and has already claimed the name to mean a particular kind of entity, which our modifiable polygons are not).

4.0 Immutable Abstractions

The solution is to model mathematical entities as immutable entities. Operations defined over immutable types are side-effect free and therefore referentially transparent (any given expression always evaluates to the same result). When an entity is immutable, it is natural for any operation to return a new entity as a result of the operation; indeed, this is what happens in many mathematical packages such as Matlab and Mathematica. For slightly more complex cases, as we shall see later, there are a few problems that must be addressed in the type system in order to make this work cleanly.

Returning to our original example of polygons and triangles,

```
class POLYGON
    n_points:INT;
    add_point(point:POINT):POLYGON
```

The add_point method now returns a new polygon object containing the additional point. It is then possible to provide a clean version of the triangle class

```
class TRIANGLE subtypes from POLYGON
     n_points:INT;
    add_point(point:POINT):POLYGON;    -- returns a polygon
```

The add_point method simply creates a new polygon which includes the additional point (in this case, it might be reasonable for it to return a RECTANGLE as well, which is still perfectly typesafe (covariant in the return type).

In terms of method postconditions, the add_point method now has the postcondition that result.n_points = initial(self.n_points)+1, which can be maintained by the add_point method of the triangle class. The important point is that, by making the postcondition say something about the return type rather than about self, the class invariant on self in triangle (n_points = 3) may be preserved.

There are many ways to implement immutable objects. Immutable objects may be implemented as actual values (primitive or composite) or as references to actual values or even as applied closures yielding actual values, but in all cases the value of the immutable object is the same and never changes for as long as it exists. In contrast, mutable objects are best used to model entities that have an identity plus a current state. The idea of an object identity bound to a modifiable state introduces side effects into the language, which can make expressions referentially opaque (an expression involving a reference object may evaluate to a different result each time that it is invoked).

5.0 Methods with Arguments

In the above discussion, only the return type needed to be specialized. What happens when the argument must also be specialized? This fre-

quently occurs in operations where, when both operands are of the same type, the result is also guaranteed to be of the same type.

Consider B which is a specialized subtype of A, such that the method m has the signature m(a:A):A in A. We then wish to support the signature m(b:B):B in the class B.

For a concrete example, consider sets which are a kind of bag, with the stronger class invariant that no element in a bag is repeated. Implementing the bag abstraction, with a couple of sample methods might look as follows (for now, we ignore the parametrization of the container class for the sake of simplicity).

```
class BAG
   union(arg:BAG):BAG;
a:BAG;  -- Contains 3
b:BAG; -- Contains 1,1,2,4
c:BAG := a.union(b);  -- c now contains 1,1,2,3,4
-- Postcondition - c.size = 5
```

In the above example, we assume that the union operation is defined to return a bag with the maximum number of occurences in either arg or self. Thus, this definition of union is consistent with the standard set-theoretic definition of union (when both self and the argument do not contain duplicates, neither does the union).

Specialize the argument and return type?

A natural solution is to attempt to support the signature m(B):B in the class B. Since the argument types are a special case of the precondition, and since B is a subtype of A, n $Pre_{m,B}$ => $Pre_{m,A}$ which is not sufficient to support substitutability.

In the case of our example involving sets:

```
class SET subtype of BAG
   union(arg:SET):SET;
```

From the point of view of substitutability, this is a non-starter. If we were to replace the BAG `a' by a SET in the example above, the union operation would have the wrong argument and return types.

Specialize only the return type?

Another choice is to avoid the typing problem by generalizing the argument type by supporting the method m(b:A):B. Though this eliminates the typing problem in the argument position, the return type of the method may no longer be sufficient. For the kinds of operations we are considering, the operation is only guaranteed to stay within the same domain if both operands are of the same type.

Returning to our example:

```
class SET subtype of BAG
   union(arg:BAG):SET;
```

This second definition of union still violates substitutability.In the above

```
s:SET := 3;
a:BAG := s;  -- Contains 3
b:BAG; -- Contains 1,1,2,4
c:BAG := a.union(b);  -- Contains 1,2,3,4
```

code, if ‘a’ were to be substituted by a SET, the result would be a set as well, and would not be able to contain any duplications of the number ‘1’. Thus, the result of using a set instead of a bag for ‘a’ will be different, and the implicit postcondition, that the number of items in the result is 5, will be violated.

More precisely, the union operation in BAG has the postcondition

$$\forall \text{ items i in self and arg, } \text{result.n\_occurs(i)} = \text{initial(max(n\_occurs(i), arg.n\_occurs(i))} \quad \textbf{(EQ 1)}$$

The postcondition in SET, however is

$$\forall \text{ items i in self and arg, result.n\_occurs(i)} = 1 \quad \textbf{(EQ 2)}$$

Keep the same signature?

A final solution is to avoid the method specialization altogether. Thus, we may support the method m(b:A):A in the class B. Clearly, this causes no problems with subtyping, since both argument and return types are invariant.

We can see this in the case of the example:

```
class SET subtype of BAG
   union(arg:BAG):BAG;
```

In this case substitutability is not violated. However, a more serious problem is introduced. It becomes impossible to stay within a domain without constantly slipping into weaker and weaker supertypes.

```
a:SET; -- contains 1,2
b:SET; -- contains 3
c:BAG := a.union(b);
```

As may be seen, even though we can guarantee that the result of the union operation will be a SET, the return type of the signature is a BAG. *In practice, this weaking is completely unacceptable*. It means that we cannot operate on sets cleanly - we keep getting bumped up to a higher level of abstraction, when we are certain that the result *must* be a set.

A further disadvantage of this approach is that it is harder to make use of more efficient algorithms that may be available to perform the same operation.

6.0 Static Covariance: The Overloading Rule

What we need is to be able to choose the right method based on both the type of self and the type of the argument i.e. a multi-method. With multi-methods, the interface to the SET class may contain two separate methods,

one to handle the general case of a union with bags, and the other to handle the more specialized case of a union of a set with another set.

```
class SET subtype of BAG
   union(arg;BAG):BAG;
   union(arg:SET):SET;
```

Languages such as CLOS [BK88] and Cecil [Cha93] permit multi-methods which dispatch on more than one argument. This is a viable but expensive solution; multi-method dispatch is inherently considerably more complex than singly dispatched methods. Though vigorous type-inference might eliminate some of these costs, this expense was not a viable design choice for a high-performance language such as Sather.

Fortunately, multi-methods are not required, since the choice of method may be made statically through the use of overloading. Thus, the solution is to support both methods in the interface of B, `m(a:A):A` as well as `m(b:B):B`. The choice of method is determined at compile-time, based on the declared type of the argument.

In addition to the benefit of efficiency, with overloading the choice of method is changed from a dynamic decision to a static one, permitting compile-time type checking.

Note that the implementation of the more general union operation could be written using the more specific method as:

```
union(arg:BAG):BAG is
   typecase arg
   when SET then return union(arg);  -- Uses the second union method
   else
      -- perform the more general bag union
   end;
end;
```

6.1 The Overloading Rule

The minimum degree of overloading that must be permitted to support the above usage is determined by the nature of specialization. The nature and design rationale behind the Sather overloading rule is described in a related report [GSW97].

In summary, two methods are permitted to overload iff there is a subtyping relationship between every pair of corresponding argument types. Differences in the return type are not used in determining overloading. At the point of call, the most specific method that matches is chosen; it is an error if there is more than one most specific method. In the example above, the two version of the `union` operator take arguments of `SET` and `BAG` respectively. Since `SET` subtypes from `BAG`, the overloading is permitted

The design of the Sather overloading rule is complicated by the presence of supertyping in the language; other languages which do not support supertyping can provide a less restriced form of overloading. It is interesting, however, to note that the kind of overloading that is permissible in the

presence of supertyping is exactly that which is required to support specialization. In some ways, this restriction on overloading is desirable in any case, to prevent users from overloading methods which happen to have the same name but which are not specializations of each other.

6.2 Overloading vs. Overriding

It is also possible to over-ride an inherited method by generalizing it. The distinction lies in the nature of the arguments to the method. If the arguments to a method are more general than (supertypes of) the arguments to the inherited method, the new method, being more widely applicable, over-rides the inherited method. If the arguments are specialized, then overloading occurs, provided that the methods can co-exist in the interface according to the overloading rule.

Another way of looking at this distinction is in terms of pre conditions. Generalization, or over-riding of a method occurs when the method preconditions become less restrictive (contravariant). Specialization by overloading of a method should be used when the method preconditions become more restrictive (covariant). When a method is specialized, the general version must still be made available in order to ensure substitutability.

7.0 What about Mutable Classes?

The above discussion presents a clean inheritance hierarchy provided that immutable abstractions are used. However, since immutable classes provide a copy of the class when any modification occurs, they may be considerably less efficient than their mutable counterparts. What should the interfaces of these mutable classes look like, and are the possible typing relationships between them?

We start with the mutable polygons mentioned in Section 2.0 (with the names amended to reflect their mutable semantics).

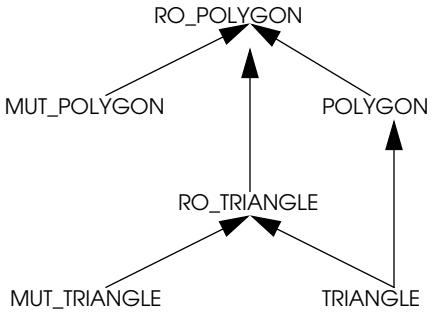
```
class MUT_POLYGON
   n_points:INT;
   add_point(point:POINT);
      postcondition result.n_points = initial(n_points)+1;

class MUT_TRIANGLE
   n_points:INT;
```

The postcondition of the mutating method add_point in MUT_POLYGON cannot be maintained in MUT_TRIANGLE, since this violates the triangle invariant that it has exactly three points. Hence, there is no subtyping rela-

tionship between mutable triangles and mutable polygons. We can, however, add non-mutating operations to the polygon and triangle interfaces.

```
class MUT_POLYGON
   n_points:INT;
   add_point(point:POINT);
   add_point(point:POINT):MUT_POLYGON;

class MUT_TRIANGLE
   n_points:INT;
   add_point(point:POING):MUT_POLYGON;
```

The immutable methods are common to both the mutable and immutable abstractions. We abstract this intersection of the two interfaces into a read-only interface. We denote these read-only abstractions with the prefix RO_. The read-only interface corresponds to a factoring out of the contravariant methods common to the mutable and immutable classes, but, more importantly, corresponds exactly to the distinction between mutable and immutable methods.



Note that the read-only interface, though it may have the same set of methods as the immutable interface, has a **different meaning from the immutable interface.** A variable which has the type of a read-only interface makes only the immutable interface visible. The object referred to by the variable, however, **may be mutable** and may be mutated through other aliases which provide the mutation-permitting interface. [DL92] show that if aliasing is prohibited, immutable types may be subtypes of mutable abstractions, since the mutating operations cannot be observbed through the immutable supertype interface.

The above diagram illustrates the potential subtyping relationships between the various abstractions. Not all of these types or subtyping relations need be represented in the type system. Furthermore, type relations between mutable classes may also be legal, provided the subtype preserves the class invariant. Since there are no direct subtyping relationships between mutable polygons and triangles, any application that seeks to exploit the relationship between triangles and polygons must make use of the read-only interface, which provides all the immutable operations only i.e. all the operations that may be safely used on polygons, even when they are substituted by triangles.

7.1 The Value of a Mutable Object

It is possible to take an immutable snapshot of an object at any particular point in time, and this is the "value" of the object at that particular instant. Thus, all our mutable interface provide the methods such as MUT_TRIANGLE::value:TRIANGLE. This method provides a conversion from an mutable to an immutable object.

8.0 Object Equality

In the context of mutable objects, the nature of equality may sometimes get confusing - is it the equality of the object pointers or the equality of the contents? Some languages provide several levels of equality (the famous eq, eql and equal), frequently a source of confusion to beginning programmers. The theoretical aspects of equality relations are dealt with in [Cas95]. We merely point out that the notion of immutable object equality may be used to guide our notion of mutable object equality.

In the mathematical world, this confusion does not arise: two objects that have the same set of values (two triangles with the same coordinates, for instance) are equivalent in all respects and therefore equal.

The immutable definition of equality preserves the substitutability principle - if two supertype objects are equal, substituted subtype objects must also be equal. The clean definition of equality in the case of immutable objects can be used to define the equality of mutable objects - two mutable objects are considered equal at any time if their value is equal. Two objects references are equal if the objects they point to return values that are equal. Thus, the equality of two reference objects is defined in terms of the equality of the corresponding immutable objects at that time, which includes all of their contained state.

9.0 The Cost of Immutability: Freezing

Given the above discussion, it is clearly cleaner and safer (immutable objects do not suffer from bugs caused by aliasing) to use immutable objects in many contexts. The main problem with immutable objects is the inordinate cost involved in all modification operations.In this section, we mention one simple way to avoid much of this overhead, which is actually used in the standard Sather library.

The cost savings is based on the observation that it is a fairly common programming practice to make use of the modification operations when setting up a data structure and to never modify the data structure afterwards it has been created. In the case of a graph, for instance, it may be convenient to create an empty graph and then add nodes and edges until it assumes the desired structure. From that point on, the structure may never be modified. This notion can be captured by the notion of freezing mutable object:

- Freezing a mutable object sets a boolean in the object after which no further modifying operations are permitted. The "is_frozen" flag is

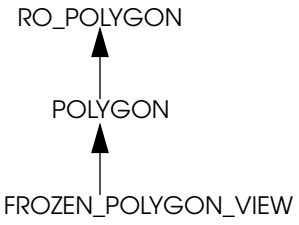
checked in the precondition of all mutating operations. This concept is used in other libraries such as JGL [Gla97].

- Frozen views are adaptor classes that take a frozen mutable class and provide an immutable wrapper.

```
class MUT_POLYGON_IMPL

   readonly attr is_frozen:BOOL;          -- Set initially to false
   freeze is is_frozen := true; end;

   add_point(p:POINT)
      precondition ~is_frozen
   is...

   value:POLYGON  is
      if is_frozen then
         return FROZEN_POLYGON_VIEW::create(self);
      else
         return  POLYGON::create(points);
      end;
```

The adapter that present frozen polygons as immutable is shown below :

```
class FROZEN_POLYGON_VIEW
   subtypes from POLYGON
   private attr from:MUT_POLYGON;
    -- Delegate calls to "from"

   create(from:MUT_POLYGON):SAME
      precondition from.is_frozen
   is
```



All calls on the adaptor are delegated to the private attribute from.

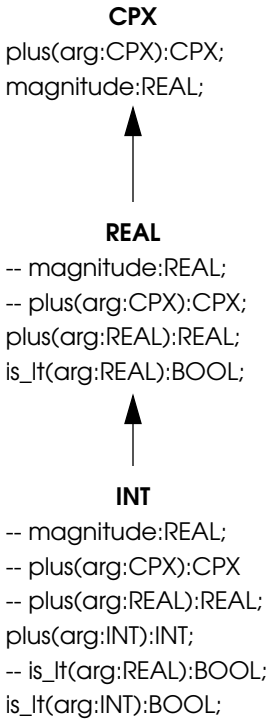
There are a few points to note about freezing

- Freezing is cheap - it only involves setting a boolean variable, and the precondition checks may be eliminated in debugged production code.
- Freezing is one-way - an object once frozen may never be unfrozen. This is critical to the immutable semantics.

Freezing is similar to the use of a mutable class through its read-only interface. However, while using a frozen class will guarantee immutability, using the read-only interface will result in errors if the original object is modified through aliases unless the aliasing is restricted.

10.0 Numbers - Complex, Real and Integer

Using value semantics (as is usually done with these classes in any case) we can conveniently model the basic number hierarchy in the conventional mathematical manner.



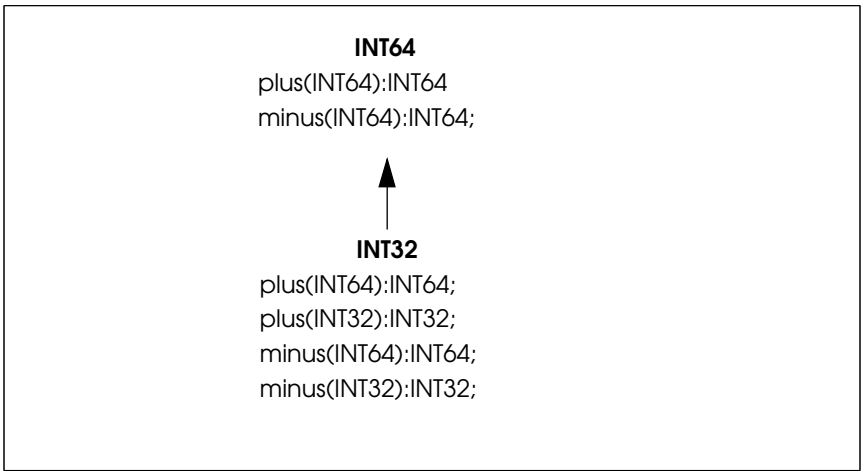
The above hierarchy demonstrates how the overloading rule may be used to obtain clean subtyping relations in the presence of specialization.

10.1 64 and 32 bit numbers

In [LM95] it is claimed that "smaller integers cannot be a subtype of larger integers because of observable differences in behavior; for example, an overflow exception that would occur when adding two 32-bit integers would not occur if they were 64-bit integers".

The substitutability of a 32 bit integer type for a 64 bit integer type actually depends on the exact nature of the class invariant that must be preserved. For instance, if we take a 64 bit integer abstraction to embody the invariant that its value < 2^64, and a 32 bit integer to embody the invariant

that its value < 2^32, then the invariant of the 32 bit integers certainly implies the invariant of the 64 bit integer and we may have



The 32 bit integer class provides specialized 32 bit methods for addition and subtraction, in addition to the general methods provided in the 64 bit class. *Note that the substitutability is safe, in terms of behavior, overflow exceptions and in all other respects.*

We could define the 64 bit integer class in such a way that substitutability is impossible, for instance, with the invariant that it has exactly 64 bits. Then the method INT64::does_not_have_64_bits[1] cannot be properly subtyped in INT32. Then, INT32 clearly violates this invariant, and subtyping is not possible. This is, however, an intentionally perverse definition of INT64, which specifically prohibits certain kinds of subtyping.

The same reasoning holds for the FAT_SET example presented in [LM95]. Unsurprisingly, if the invariant of a class is that elements may never be removed, then it is not possible to substitute it with a class that violates precisely that invariant. By design of an abstract type, it is possible to proscribe certain kinds of subtyping.

11.0 Correct Method Naming

Methods that behave differently must have different names. This may seem like an obvious point, but it is one that is easy to violate if method signatures alone, and not the method semantics, are taken into account. In other words, type signatures are not everything. [LM95] makes this point by considering other aspects of the class semantics via the notion of constraints on behavior.

1. INT64::does_not_have_64_bits:BOOL will return false, while INT32::does_not_have_64_bits:BOOL will return true, thus violating the upward implication of preconditions required for substitutability.

We illustrate the point with the case of the ‘insert’ method in sets and in bags. For example, it seems reasonable to support a BAG::insert which is then specialized in the SET abstraction.

```
class BAG{T}
   insert(element:T):BAG{T};

class SET{T}  subtype of BAG{T}
   insert(element:T):SET{T};
```

There is no problem with the typing of the above methods. The return type of the SET::insert is specialized, and is therefore conformant to BAG::insert. However, the second method is not substitutable for the first. Consider the post conditions of the methods above:

```
class BAG{T}
   insert(e:T):BAG{T};
      postcondition  result.size = initial(size)+1;

class SET{T}  subtype of BAG{T}
   insert(e:T):SET{T};
      postcondition initial(contains(e)) and result.size = initial(size)
         or result.size = initial(size)+1;
```

The postcondition in the case of the Set states that the resulting size remains the same if the set already contained the element, otherwise the resulting size is increased by 1. The crucial test for substitutability is whether the postcondition of SET::insert implies the postcondition of BAG::insert.

initial(contains(e)) and result.size = initial(size) or result.size = initial(size)+1

? =>

result.size = initial(size)+1;

Clearly, when the element is already in the set, the antecedent is true with the size of the result equal to the initial size, and the consequent is false. Thus the implication does not hold and SET is not substitutable for BAG as they are defined here.

The right approach is to distinguish between the two notions of insertion.

```
class BAG{T}
   append(e:T):BAG{T}
      -- add e to the bag, even if it is already present
      postcondition result.size = result.size + 1;
   insert(e:T):BAG{T}
       -- insert only if e is not already in self
      postcondition initial(has(e)) and result.size = initial(size) or
         result.size = initial(size)+1;

class SET{T} subtype of  BAG{T}
   append(e:T):BAG{T};  -- Same postcondition as BAG::append
   insert(e:T):SET{T};          -- Same postcondition as BAG::insert
```

With the above definition, the insert routine may be safely specialized - the postcondition is the same in both cases. The append routine must continue to return a BAG, since appending an element to a SET may result in the presence of duplicate elements, requiring a BAG.

Using the structure of these immutable abstractions to guide the mutable abstraction, MUT_BAG should provide both an append and an insert method, while the MUT_SET abstraction can only provide the insert method.

The Eiffel library design recommends consistent naming, which means using the “same names for all structures regardless of the semantic differences” [Mey94]. Rules such as this can end up obscuring important differences in method semantics and may promote erroneous subtyping relationships.

12.0 Related Work

This report is aimed at object-oriented practitioners; in the course of designing the Sather libraries, the problems related here arose repeatedly; indeed, the confusion is quite widespread, as we illustrate by our quotes. The problem arises principally from differences between the mathematical and the object based metaphors that underlie library design. The underlying theoretical ideas have been explored, though not, to our knowledge, in the context of actual library design.

[LW94] explicates guaranties of substitutability under subtyping, based on object behavior, and the importance of considering object protocol in addition to type signatures when determining substitutability. The distinction between specialization (using overloading) and subtyping that we draw in this report is largely a restatement of [Cas95]. Others have noted that some subtyping problems may be avoided by considering immutable data types ([Ock95], [Win97]). It has also been shown in [DL92] that subtyping between mutable and immutable types is possible if aliasing is restricted, so that an object may only be viewed via a subtype or a supertype variable, but not both at the same time. The problem has mosly been explored from the perpspective of the formal semantics of objects, rather than from the point of view of correct modelling. The formal semantics, while useful in understanding language restrictions, obscures the simple nature of the underlying modelling problem. Our primary goal was to explain, in a non-

theoretical manner, the modelling problem, why the problem arises, our solution in Sather and the implications for practical library design.

Libraries, such as the collections package in Java libraries by D. Lea [Lea97] distinguish between value and reference semantics as we advocate. However, they do not deal with the issues of subtyping and cannot make use of the covariant specialization that our overloading rule permits.

Other libraries, such as the Karla library [FNZ97] deal extensively with the problem of mutable classes. Since the is-a relationships from mathematics do not hold in the world of, for instance, mutable graphs, they have devised a generator for the combinatorial number of possible concrete classes that may arise.

13.0 Conclusions

The answer to the FAQ question 28.1 mentioned in the abstraction runs (in part) as follows:

> The sad reality is that it means your intuition is wrong. Look, I have received and answered dozens of passionate e-mail messages about this subject. I have taught it hundreds of times to thousands of software professionals all over the place. I know it goes against your intuition. But trust me; your intuition is wrong.
>
> The real problem is your intuitive notion of "kind of" doesn't match the OO notion of proper inheritance (technically called "subtyping"). The bottom line is that the derived class objects must be substitutable for the base class objects. In the case of Circle/Ellipse, the setSize(x,y) member function violates this substitutability.

While the above answer is true, it does not capture the real cause of the problem, which is the distinction between value and reference abstractions. We have described the distinction in detail, and the use of overloading in correctly modelling mathematical entities using value abstractions. We also describe how these clean mathematical abstractions may be used to guide the design of the more efficient, mutable abstractions.

Some of the lessons we draw for library design:

- Pay close attention to the underlying object metaphor of the domain being modelled. If the metaphor used when modelling is different from the original domain metaphor, then entailments from the original domain will not hold in the modelled domain.
- Clarity is important; terms from the world of mathematics (such as polygon, graphs, sets etc.) should not be used to name classes that model entities that are subtly different. Inferences from the mathematical domain may not hold in this modelled domain, and this should be made clear to clients of the class (and, often, to the class designer as well!).
- When type-safe substitutability is possible, overloading may be necessary to permit specialization of operations. In general, contravariance

(or invariance) of the preconditions only causes problems when there is some underlying problem with the substitutability relation.

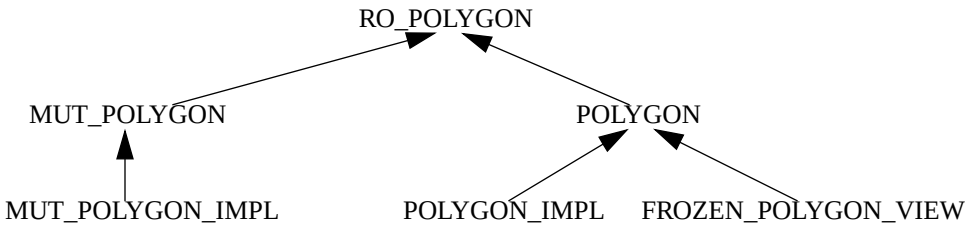
- Signatures are not everything; methods with different semantics must be given different names, even if they happen to have conforming signatures.

Appendix A Mutable, Immutable and Frozen Polygons

The complete code for mutable, immutable and frozen polygons is shown below. We first present the abstract classes.

```
class RO_POLYGON
   is_frozen:BOOL;
   value:POLYGON;
   n_points:INT;
   add_point(p:POINT):SAME;

class MUT_POLYGON  subtypes from RO_POLYGON
   add_point(p:POINT);

class POLYGON subtypes from RO_POLYGON
-- same interface as RO_POLYGON
```

The typing relationships are as shown below.



Note that the leaves of the type graph are implementation classes, while all interior nodes are abstract.

The mutable polygon may then be defined as follows

```
class MUT_POLYGON
   private attr points: ARRAY of POINT;
   readonly attr is_frozen:BOOL;

   create:SAME is
      res:SAME := new;
      res.points := new ARRAY of POINT;
      res.is_frozen := false;
      return res;
   end;

   add_point(p:POINT):MUT_POLYGON is
      res:SAME := MUT_POLYGON::create;
      for old_point:POINT  in points
         res.points.append(old_point);
      return res;
   end;

   add_point(p:POINT)  precondition ~is_frozen  is
         points.append(p);
    end;

   freeze is  is_frozen := true; end;

   value:POINT is
      if is_frozen then  return FROZEN_POLYGON_VIEW::create(self);
      else return POLYGON(self); end;
   end;

end;
```

References

[BK88] Daniel G. Bobrow and Gregor Kiczales. **Common LISP object system specification.** Technical Report 89-003, MOP Draft number 10, MIT, December 1988.

[Cas95] Guiseppe Castagna. **Covariance and contravariance: Conflict without a cause.** *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, March 1995.

[Cha93] Craig Chambers. **The cecil language: Specification and rationale.** Technical report, University of Washington, March 1993.

[DL92] Krishna K. Dhara and Gary T. Leavens. **Subtyping for mutable types in object-oriented languages.** Technical Report 92-36, Iowa State University, November 1992.

[FNZ97] Jozsef Frigo, Rainer Neumann, and Wolf Zimmermann. **Mechanical generation of robust class hierarchies.** In *TOOLS97*, 1997.

[Gla97] G. Glass. **The Java Generic Library.** *C++ Report*, 9(1):70–74, January 1997.

[GSW97] Benedict Gomes, David Stoutamire, and Boris Weissman. **The overloading rule in Sather.** Technical Report Unknown, International Computer Science Institute, July 1997.

[Lea97] Doug Lea. **Overview of the collections package.** http://gee.cs.oswego.edu/dl/classes/collections/index.html, 1997.

[Lis88] Barbara Liskov. **Data abstraction and hierarchy.** *SIGPLAN Notices*, 23(5), may 1988.

[LW94] Barbara Liskov and Jeannette Wing. **A behavioral notion of subtyping.** *ACM Transactions on Programming Languages and Systemsn*, November 1994.

[Mar96] Robert C. Martin. **The Liskov substitution principle.** *The C++ Report*, March 1996. http://www.sigs.com/publications/docs/cppr/9603/cppr9603.c.martin.html.

[Mey94] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[Mey96] Bertrand Meyer. **Static typing and other mysteries of life.** *Object Currents*, 1(1), January 1996. http://www.sigs.com/publications/docs/oc/9601/oc9601.c.meyer.html.

[Ock95] John Ockerbloom. **Exploiting structured data in wide-area information systems.** Technical Report CMU-CS-95-184, Carnegie Mellon University, 1995.

[Sha96] David Shang. **Are cows animals.** *Object Currents*, 1(1), January 1996. http://www.sigs.com/publications/docs/oc/9601/oc9601.c.shang.html.

[Win97] Jeannette M. Wing. **Subtyping for distributed object stores.** Technical Report CMU-CS-97-121, Carnegie Mellon University, April 1997.