Understanding Object-Oriented Languages: Semantics and Types *(Continued)*

Kim B. Bruce
Princeton University & Williams College

December 8, 1998

5 Adding F-bounded parametric polymorphism

One of the frustrations for Java programmers is the lack of support for parametric polymorphism. As a result, programmers create data structures holding elements of type Object (or some other type guaranteed to be a supertype of all types of elements likely to be used).

For example, suppose you want to use the built-in java.util.stack. It's declaration is similar to:

```
public class Stack extends Vector{
   public Object push(Object item){...}
   public Object pop(){...}
   public Object peek(){...}
   public boolean empty(){...}
   public int search(Object o){...}
}
```

Suppose we wish to manipulate elements of type `Point` with the stack `myStack`. Creating the stack and inserting items is no problem:

```
Stack myStack = new Stack();;
Point aPoint = new Point(2.0,3.0);
myStack.push(aPoint);
```

The use of a `Point` as the parameter to `push` is well-typed because `Point` is a subtype of `Object`.

However, suppose we wish to remove the top element from the stack and then send it a `move` message. The result of evaluating `myStack.pop()` is statically an element of type `Object`, even though we know we pushed a `Point` on the stack. Because `Object` does not contain the method `move`, we must first cast the element to type `Point` before sending the `move` message:

```
Point stackPoint = (Point)(myStack.pop()).move(1.0,2.5);
```

Of course it is even worse if we wish to put elements of type `int` on the stack, as `int` is not an extension of `Object`. To push an integer `n` on the stack, we must push on the corresponding `Integer` via `myStack.push(new Integer(n))`. Popping off such an element is also painful as you must pop it off, coerce it to `Integer`, and then send it the `intValue()` message before being able to do anything with the value.

In summary, we have two major problems with the stack implementation using `Object`:

1. We can't enforce that the stack only hold items of a fixed type.

2. Whenever we remove something from the stack, we must cast it to the type we expect it to be so that we can send messages to it.

These two problems result in ugly code with lots of casts and leave the code open to errors if somehow an element of some other type is pushed onto the stack.

The Java extension, GJ, attempts to remedy this by adding parametric polymorphism to Java. In GJ we can rewrite Stack as:

```
public class Stack<Elt> extends Vector<Elt>{
   public Object push(Elt item){...}
   public Elt pop(){...}
   public Elt peek(){...}
   public boolean empty(){...}
   public int search(Elt o){...}
}
```

We can create and add an element to the stack as follows:

```
Stack<Point> myStack = new Stack<Point>();;
Point aPoint = new Point(2.0,3.0);
myStack.push(aPoint);
```

Pushing anything other than a `Point` onto the stack results in a static type error.

The code for popping a point off the stack and moving it is now much clearer, as we don't need casts.

```
Point stackPoint = myStack.pop().move(1.0,2.5);
```

GJ can also handle more sophisticated examples of parameterized classes in which we wish to restrict the collection of type parameters which can be used.

Suppose we wish to implement a class representing ordered lists. In order to insert an element, we must compare it with other elements in the list. As a result, we can only insert elements of classes which provide the appropriate methods comparing elements:

```
interface Comparable {
   public boolean equal(Comparable other);
   public boolean greaterThan(Comparable other);
   public boolean lessThan(Comparable other);
}
```

We can then write a class `OrderedList` as follows:

```
public class OrderedList<Elt implements Comparable> extends ... {
   public void insert(Elt item){...}
   public Elt removeFirst(){...}
   public boolean empty(){...}
   public int search(Elt o){...}
```

Because `Elt` is constrained to implement `Comparable`, we can send messages `equal`, `greaterThan`, and `lessThan` to terms of type `Elt`.

Now we can instantiate `OrderedList` with any class which implements `Comparable`. This sounds fine, but we soon discover that it is hard to write a class implementing `Comparable`. The difficulty is that the comparison operators have to take arguments of any other class implementing `Comparable`. Here is an example of the difficulties:

```
public class Simple implements Comparable {
   protected int x = 0;
   ...
   public boolean greaterThan(Comparable other){
      if (other instanceof Simple){
         return (x > ((Simple)other).x)
      } else { ... raise an exception??...}
   }
   ...
}
```

The code for `greaterThan` is now much more complex than expected. We must test if `other` is of type `Simple`, and then cast it to `Simple` before being able to do anything with it. And what do you do if the `instanceof` test fails?

Moreover, it is pretty clear that you could virtually never use a pre-existing class to instantiate `OrderedList`, because the comparison methods of existing classes are highly unlikely to take parameters of type `Comparable`. Instead they most likely will take parameters of the same type as the class being defined.

For example, `Simple` would normally be defined instead as `SimpleC`, below:

```
public class SimpleC implements Comparable {
   protected int x = 0;
   ...
   public boolean greaterThan(SimpleC other){
        return (x > other.x)
      }
   }
   ...
}
```

The bounded polymorphism illustrated above was invented in the mid-80's. However, by the end of the '80's, examples like the above showed that regular bounded quantification was not expressive enough to capture the restrictions on type parameters needed for many important examples which contained what came to be known as "binary methods". These are methods like `equal`, `greaterThan`, and `lessThan` in which the parameter types are expected to be the same as the class being defined (as they are in `SimpleC` above).

A technique known as F-bounded quantification was invented to handle these more complex cases. We illustrate the technique by example. Rather than defining the interface `Comparable`, we now define a parameterized interface, `ComparableF`:

```
interface ComparableF<T> {
   public boolean equal(T other);
```

```
   public boolean greaterThan(T other);
   public boolean lessThan(T other);
}
```

Any class C implementing ComparableF<Elt>, for type Elt must have methods equal, greaterThan, and lessThan which take parameters of type Elt.

Notice that the class SimpleC above implements ComparableF<SimpleC>. In general, if class Elt implements ComparableF<Elt> then it must support comparison operations which take parameters of type Elt. This is a sort of recursive constraint in that the upper bound of Elt involves Elt, but there is no real difficulty in figuring out if an existing class satisfies such a constraint. You need only check that it has methods of the appropriate types.

We now can rewrite class OrderedList as:

```
public class BPOrderedList<Elt implements ComparableF<Elt>> extends ... {
   public void insert(Elt item){...}
   public Elt removeFirst(){...}
   public boolean empty(){...}
   public int search(Elt o){...}
}
```

The constraint on Elt is now much harder to read, but it does state exactly the constraints necessary to use BPOrderedList.

Because SimpleC satisfies these constraints we can instantiate OrderedList with SimpleC:

```
BPOrderedList<SimpleC> ordListSC = new BPOrderedList<SimpleC>();
SimpleC a = new SimpleC();
ordListSC.insert(a);
...
```

In summary, GJ's addition of F-bounded polymorphism to Java has greatly increased the expressiveness of Java's type system. Whereas before we had to use Object or another fixed type to represent a whole collection of possible types which could be used in a data structure, we can now instantiate the class to a specific type as long as it satisfies the given constraints.

Unfortunately, F-bounded quantification does not interact well with the class or subtype hierarchies in object-oriented languages. The problem is that if a class satisfies an F-bounded constraint, its subclasses will not satisfy the constraint. For example, define:

```
public class ComplexC extends SimpleC {
   ...
   public boolean greaterThan(SimpleC other){
        return (x > other.x) && ...
      }
   }
   ...
}
```

No matter how few additions or changes are made in the subclass (and even if no methods from the subclass are overridden), ComplexC will implement ComparableF(SimpleC), not ComparableF(ComplexC). As a result it cannot be used with BPOrderedList.

The difficulty is that we are not allowed to change the types of method parameters in subclasses. As a result, subclasses conflict with the use of F-bounded polymorphism. We will see later on that the use of `MyType` and match-bounded polymorphism gives similar expressiveness as F-bounded polymorphism, but *is* consistent with the subclass relation.

Several compromises were made in order to get GJ programs to run with the current crop of virtual machines (which do not support polymorphic classes). The implementation of parametric classes with GJ essentially proceeds by translating a parametric class into a standard Java class in which the type parameters are erased and replaced by the upper bound of the type parameter (or `Object` if there is no upper bound). Type casts are then automatically inserted in several places in the translated program in order for the resulting Java program to be type-safe. While all of the casts are guaranteed to succeed, the result is that the translated program ends up looking very similar to the way one would normally write the code in plain Java. Moreover, even though the casts are guaranteed to succeed, they will still be checked at run-time, slowing down the execution of the program.

Other complications arose in implementing GJ, especially with attempts to deal with arrays. I urge you to read the on-line tutorial at the GJ web site (see the link from the class handouts page) in order to get more details on GJ.

In summary, GJ's addition of F-bounded polymorphism to Java helps the programmer to express type constraints in a readable and type-safe way. While the implementation on current virtual machines will not run any faster than the usual ways of writing generic code in Java, future enhancements to the virtual machine should provide for fast, safe execution of GJ programs.

6 More (too) flexible typing with Eiffel

The programming language Eiffel was designed in the mid-80's by Bertrand Russell. It is a statically-typed class-based object-oriented language which supports multiple inheritance, bounded polymorphism, and executable assertions. Like Java, all objects are implicit references and garbage collection is built in.

Here is a portion of a sample Eiffel class.

```
   class RATIONAL

   inherit
      ORDEREDPAIR [INTEGER]
        rename  x as n, -- can change names of features
                y as d
        redefine same   -- indicates that same will be redefined
        end
      OTHERCLASS
        redefine ...
        end

   creation
      CreateRat

feature{ALL}
```

```
 error: boolean;

 CreateRat is
         -- create a rational
     do
         d := 1
     end; -- Create

   set(numer, denom : INTEGER) is
         -- set the numerator and denominator
      require -- precondition
         denom /= 0
      local
         gcd : INTEGER
      do

         n := numer;
         d := denom;

         if d < 0 then
             n := -n;
             d := -d
         end;

         gcd := reduce;

         n := n // gcd;
         d := d // gcd
      ensure  -- postcondition
         d > 0
      end; -- set

   plus(other : like Current) : like Current is
      local
         sumnum, sumden : INTEGER;
      do
         sumnum := n*other.d + other.n*d;
         sumden := d*other.d;
         !!Result.Create;
         Result.set(sumnum,sumden)
      end; -- plus

   ...

feature {NONE}  -- private method
   reduce : INTEGER is   featureALL
end
```

```
invariant d /= 0
```

The above example illustrates several features of Eiffel. First it is possible for a class to inherit from multiple classes. Methods in inherited classes can be renamed either for convenience or to resolve name clashes between classes inherited from. Methods that are overridden must be declared to be "redefined". This is required to avoid the problem of having a programmer accidentally override a method in a superclass. It is also possible to change the visibility of inherited methods with a declaration in the `inherit` clause, though we don't show it here.

Creation routines may be used as constructors. If `r` has type `RATIONAL`, then `!!r.CreateRat` will create a new object of type `RATIONAL`. Instance variables not explicitly initialized are given a default value (e.g., 0 or `void` - which corresponds to `null` in Java).

Features include both instance variables and methods. The `feature` clause indicates the visibility of items listed. The default is `ALL`, `NONE` corresponds to Java's private, while you can also list any collection of classes (e.g. "friends") which you wish to be able to have access to particular features. Instance variables are only available as read-only outside of the methods of the class. In fact, from outside the object it is impossible to distinguish between instance variables and parameterless procedures.

In method `set`, the precondition is given using keyword `require` and the postcondition uses `ensure`. The class invariant at the bottom of the class is preceded by `invariant`. It is also possible to specify loop invariants and "variants". A variant is an integer expression which is always positive, but which gets closer to zero after each iteration. It is used to help prove termination of the loop. The assertions can be labeled in order to give good error messages when an assertion fails. Determination of which kinds of assertions are checked at run-time is made by setting a compile-time flag. The default is the pre-conditions are always checked, but others are not.

`Current` is used in Eiffel as the name of the object executing the method rather than Java and C++'s `this` or Smalltalk's `self`. If `id` is a declared identifier, then `like id` can be used to stand for the static type of `id`. Thus the method `plus` in `RATIONAL` uses `like Current` as the type of the argument and result of the method. If `plus` is sent to an object of (static) type `RATIONAL`, then its actual parameter must also be of type `RATIONAL`, and the result will be of type `RATIONAL`.

If `plus` is inherited in a subclass, `SUBRATIONAL` of `RATIONAL`, then the meaning of `like Current` will change appropriately. That is, if `plus` is sent to an object of (static) type `SUBRATIONAL`, then its actual parameter must also be of type `SUBRATIONAL`, and the result will be of type `SUBRATIONAL`.

This is very handy and provides great flexibility in creating subclasses. For example, here are two classes providing for singly and doubly-linked nodes.The following are slightly simplified examples from the Eiffel structure library. They represent singly and doubly-linked nodes.

```
class LINKABLE [G]

feature

        item: G;

        right: like Current;  -- Right neighbor

        putRight (other: like Current) is
        -- Put ‘other’ to the right of current cell.
                do
```

```
                right := other
        ensure
                chained: right = other
        end;

end -- class LINKABLE
```

Now define the subclass:

```
class BILINKABLE [G] inherit

    LINKABLE [G]
        redefine
                putRight
        end

feature -- Access

    left: like Current;
                -- Left neighbor

    putRight (other: like Current) is
                -- Put ‘other’ to the right of current cell.
        do
                right := other;
                if (other /= Void) then
                        other.simplePutLeft (Current)
                end
        end;

    putLeft (other: like Current) is
                -- Put ‘other’ to the left of current cell.
        do
                left := other;
                if (other /= Void) then
                        other.simplePutRight (Current)
                end
        ensure
                chained: left = other
        end;

    simplePutRight (other: like Current) is
                -- set ‘right’ to ‘other’
        do
                 right := other
        end;

    simplePutLeft (other: like Current) is
```

```
                        -- set ‘left’ to ‘other’ is
                do
                        left := other
                end;

invariant

        rightSymmetry:
                (right /= Void) implies (right.left = Current);
        leftSymmetry:
                (left /= Void) implies (left.right = Current)

end -- class BILINKABLE
```

This is something that seems very handy, as it is frustrating not to be able to write a doubly-linked node as a subclass of a singly-linked node is languages like Java and C++. But now suppose we write the following routine

```
trouble(p, q : LINKABLE [RATIONAL] ) is
   do
      p.putRight(q);
      ....
   end
```

and suppose we have `sNode : LINKABLE [RATIONAL]` and `biNode: BILINKABLE [RATIONAL]`. What happens if we write:

```
trouble(biNode,sNode)
```

If `BILINKABLE [RATIONAL]` is subtype of `LINKABLE [RATIONAL]`, then this should work, instead, BANG!!!!! - we get a system crash when the code for `putRight` in `BILINKABLE` attempts to send the message `simplePutLeft` to `sNode`.

Thus Eiffel is not statically type-safe. In fact, Eiffel allows a number of statically unsafe changes in creating subclasses. One is allowed to replace the type of an instance variable, method parameter, or method result type by a subtype of the one occurring in the superclass. We have seen earlier that only method result types can be changed in a covariant way if one wishes to guarantee type safety. Eiffel also allows programmers to hide inherited methods in a subclass that were visible in the superclass, creating yet another type insecurity.

This type insecurity was first pointed out in print in 1989 by William Cook. Bertrand Meyer has had various responses over time. He has consistently claimed that the insecurity is inconsequential because Eiffel users do not write code that would result in this error. That is, while they use features like these, they automatically use them correctly. He also claimed that proposed fixes like eliminating `like` declarations and instead using bounded polymorphism forced programmers to “plan ahead” too much and would result in programmers having to go back and rewrite superclasses.

After a while of hearing further criticism of this, he then proposed a “system validity check” to ensure that these errors did not occur at run-time. Essentially the idea was to perform a conservative dataflow analysis at link-time to make sure no unfortunate combinations of types would arise at run-time due to interactions of the various classes. None of the Eiffel compiler suppliers ever implemented this check. It is not known why they did not implement it, but it is likely that either the test was too expensive to run or it ruled out too many useful programs.

Even if this test had worked, programs would have been quite fragile with respect to the test. A program could pass the system validity check, but then the addition of a new class could cause the system validity check to fail due to an unexpected interaction that was only barely related to the new class.

Three years ago, Meyer proposed an alternative “fix” to Eiffel’s static type safety problems called “No polymorphic cat-calls”. A call is polymorphic if its target can represent objects of more than one type. According to Meyer: “A routine is a CAT (Changing Availability or Type) if some redefinition changes its export status or the type of one of its arguments. A call is a catcall if some redefinition of the routine would make it invalid because of a change of export status or argument type.” His proposal is to ban “catcalls” to terms which can represent objects of more than one type. Essentially this requires that if the type of a method parameter is changed in a subclass, then you must know the exact type of any object which is a target of the corresponding message.

Meyer announced this fix with great fan-fare in an invited talk to OOPSLA 3 years ago. Three days later I found a remaining type insecurity in the language relating to changing the types of instance variables in subclasses. He has since fixed that problem. I do not know if his current proposal results in a statically type-safe language. However, I believe that his proposal is so restrictive that programmers will not be able to write useful programs that both use the dangerous features and pass his test. To this date, no Eiffel compiler implements this proposal.

In spite of its static typing problems, Eiffel was quite an advance over existing object-oriented languages. Its “design-by-contract” philosophy, which encourages the use of pre-conditions and post-conditions may be its most lasting legacy. Like Smalltalk, which is dynamically typed, it also showed how useful certain constructs could be that were not available in other statically-typed languages.

Eiffel had a chance to emerge as an important object-oriented language when experienced object-oriented programmers started losing faith in C++ as a tool to build reliable complex systems, but its type problems and slow compilers and compiled code worked against it. Ironically it lost to Java which, at least in the early days, ran significantly slower than Eiffel.

Eiffel was also one of the first commercially available compilers for object-oriented languages which supported bounded polymorphism. Here is an example.

```
deferred class COMPARABLE -- deferred is the same as abstract in Java
feature specification
    lessThan (other: like Current): BOOLEAN is
        deferred;

    greaterThan (other: like Current): BOOLEAN is
        deferred;
end -- class COMPARABLE
```

Now define:

```
class INTORD inherit COMPARABLE
feature
    value:INTEGER;
    lessThan (other:like Current) is
        do
            Result := value < other.value
        end;
```

```
    ...
end -- class INTORD
```

Clearly INTORD implements COMPARABLE. We can use COMPARABLE as an upper bound for polymorphic classes:

```
class Sorting[T -> COMPARABLE]
feature
    sort(thearray:ARRAY[T]):ARRAY[T] is
        local  ....
        do
            ......
            .... if thearray.item(i).lessThan(thearray.item(j)) ....
        end;
```

The notation indicates that the type variable T must be a subclass of COMPARABLE. Notice how much simpler this notation is than the notation for F-bounded quantification.

In summary, while Eiffel's static type system does not catch all type errors, it does suggest constructs that would be quite useful in other statically-typed object-oriented languages. In particular, the use of `like Current` seems to add a great deal of expressiveness to the language. In the next section we investigate the possibility of adding a similar construct in a type-safe way.