Inheritance, Generics, and Binary Methods in Java

M. L. Barrón-Estrada, R. Stansifer

Departamento de Sistemas y Computación
Instituto Tenológico de Culiacán
Av. Juan de Dios Bátiz s/n. Col. Guadalupe
Culiacán, Sin. CP. 80220 Tel. 667-7133804
mbarron@fit.edu,

Computer Science Department
Florida Institute of Technology
University Blvd. Melbourne, FL 32901
Phone 321-6747156 Fax. 321-6747046
ryan@cs.fit.edu

ABSTRACT

Java has adopted a mechanism to support parameterized types that will be available in the next major release. A draft specification to add generics to the Java™ Programming Language was published two years ago [1] and a new version of it in June 23, 2003 [2]. An extension of the type system based on F-bounded quantification is proposed.
A binary method is a method that has one or more parameters of the same type as the object that receives the message. F-Bounded polymorphism and binary methods can't be combined smoothly in object-oriented languages with nominal subtyping and single dispatch [3].
In this paper, we address some problems that can arise when binary methods are needed in parameterized classes and interfaces in the implementation of a Java program.

Keywords:
Binary methods, Inheritance, Java, Parameterized types.

1. INTRODUCTION

The addition of parametric polymorphism to the Java Programming Language has been under consideration for several years. Many proposals were presented and finally a draft specification to add generics to the Java programming language [1] was released in April 2001. This specification is an evolution of an early proposal called Generic Java (GJ), which is described in [4]. A prototype implementation of the Java compiler that supports generics, as described in the draft specification, is available for developers who want to start writing generic code.

Java is being updated to incorporate some new elements in the language like parameterized types and type variables to support the creation of generic code (classes, interfaces, and methods). The requirements specify that no changes to the JVM should be done to support these new features.
The technique used to translate generic code into Java bytecode is called *type erasure*. In this technique, all the type parameters defined in the parameterized type are replaced by type Object in order to make them compatible with existing class libraries. When a client uses a parameterized type to define instances of it, in the translation process, the compiler will insert some bridge methods and coercions, which are guaranteed not to fail at execution time.

In Java when the generic idiom is used to implement generic code some coercion operations are required in client code to ensure type safety. These operations increase the execution time and the class file size. The translation approach used in Generic Java to translate parameterized types improves neither performance nor class file size. When a parameterized type is translated the class file obtained is exactly the same as the one implementing the generic idiom. Client code does not explicitly require coercions but they are automatically inserted in the translation process increasing execution time and class file size. If performance efficiency important, we could get rid of cast operations in client code using a hand specialized code[1] for each particular type. Another fact of this approach is that primitive types cannot be used to create instances of parameterized types because they are not derived from type Object. The translation of parameterized types erases type parameters, which makes the types not available at run-time.

Generic code can be unconstrained or constrained. There are several mechanisms to constraint polymorphism object-oriented languages. The one proposed by Cardelli and Wegner [5] allows expressing the idea that a function can be applied to all types that are a subtype of another. This mechanism is not powerful enough to express all kind of constraints like the ones needed for recursive type definition. It is possible to define type parameters with recursive constraints using F-bounded quantification [6].

The implementation of generics in Java as described in the draft specification document is based on the F-bounded mechanism. When a type is defined to be constrained recursively, it is not possible to reuse code through inheritance because subclasses can not redefine recursive bounds to themselves and so they can not be used create instances of a parameterized type. They inherit bound defined in the superclass. A classical example that needs a recursive bound definition on the type parameter the implementation of binary methods. A recursive bound needed in order to constraint the type of the object used the binary method otherwise the programmer has responsibility to write code to verify the type of that object.

Code reuse is one of the main promises of object-oriented languages. In Java it seems that the use of F-bounded polymorphism in the definition of a class disallows the use of inheritance to generate subclasses that can be used the instantiation of parameterized types. An alternative method must be used in order to be able to create subclasses that define their own bounds allowing them

[1] This approach has also some drawbacks that we do not address here.

reuse some of the code defined in the superclass and use the subclass to create instances of parameterized types.

The purpose of this paper is to show some examples that explore the use of constrained generics in the presence of binary methods. We address the situation when a programmer needs to create a subclass from a class that has an F-bound constraint and wants to use this subclass create an instance of a parameterized type. We also show some problems that can arise in execution time.

Section 2 introduces some terms used in the paper. Section 3 states the problem we use as example in this paper and briefly describes three different implementations. Section 4 presents the implementations in detail as well as some of their problems. Section 5 examines other implementation alternatives. Conclusions are presented in section 6.

TERMINOLOGY

Binary method. A Binary method is one that contains one more parameters of the same type of the object that receives the message.
Bounded polymorphism. Bounded type abstraction. It allows one to define a function which works for every type that is a subtype of a bound *A*, and whose result type depends on *A'*.
Contravariance. Contravariance is a relationship that captures the subtyping relation. It means that changes of a particular type are opposite to the type hierarchy.
Constrained genericity. The types used as type parameters are restricted by some other types.
Covariance. Covariance is a relationship that characterizes code specialization. Changes of a particular type are parallel to the type hierarchy.
F-Bounded polymorphism. Generalizes System $F_{\leq}$ extending it with recursive types.
Inclusion polymorphism. An object can be viewed as belonging to many classes.
Inheritance. Inheritance is a mechanism to derive new classes or interfaces from existing ones.
Interface. An interface declaration contains constants and abstract methods that are supported by classes that implement that interface.
Multi-methods. Multimethods are also known as generic functions. Methods can belong to more than one class and they can be dispatched based on the types of all parameters.
Multiple dispatch. In multiple dispatch the selection of the method to be executed depends on the type of all the parameters of the message not only on the receiver.
Novariance. Novariance means that types do not change in the type hierarchy.
Parametric Polymorphism. Works uniformly in a range of types. A polymorphic function has the ability to receive a type parameter that determines the type of the argument for the application of this function. [5]
Single dispatch. In single dispatch the selection of the method to be executed depends only on the type of the object that receives the message.
Subtyping. Subtyping allows a value of a subtype to be used anywhere a value of its supertype is expected.
Unconstrained genericity. Any type can be used as type parameter in the instantiation of a parameterized type. There is no restriction.

Java is a strongly typed object-oriented language with single dispatch. It provides inclusion polymorphism and will soon provide parametric polymorphism. The creation of collections of elements such as Class<T> where Class represents a parameterized type and T a type parameter will soon be allowed.

3. PROBLEM DESCRIPTION

Our goal is to implement an OrderedList class that can be used to create ordered lists of any collection of elements that understand the messages *eq*, *le*, and *ge*, which represent *equal*, *less-or-equal*, and *greater-or-equal* respectively. The elements of the list must be orderable so the method *member* can use the *equal* message to compare the element we are looking for with the elements of the list. The method *insert* needs to use the *le* message when looking for a place where to insert a new element in the ordered list.
We examine three ways to implement the generic class that manipulates an ordered list of elements. In all the implementations we follow the rules of the Java programming language [7] and its specification to add generics to the language known as Generic Java [2].

- **Unconstrained genericity.** The generic idiom is used to implement this solution. Section 4.1 explains why this solution in not appropriate to solve this problem.

- **Genericity with a simple bound.** This approach restricts the type of the parameters used in the instantiations of parameterized types. The parameter type must be bounded to another type to ensure that it implements some methods needed. A solution using a simple bound is presented in section 4.2.

- **Genericity with a recursive bound.** Using a recursive bound on the type parameter ensures that the required arguments are of the same type than the object that receives the message. Section 4.3 shows a solution using a recursive bound.

4. IMPLEMENTATIONS

4.1. Unconstrained genericity.

Sometimes it is possible to write generic code using **Object** to implement it. This implementation is known as using the generic idiom. For example we can implement a generic Stack class that manipulates objects or a generic List that link objects. These generic classes are not required to have a specific method. They are unconstrained. In our case, to define the OrderedList class, we need to constraint the classes to have some methods (le, ge, eq) that are needed to compare elements.
Object contains the definition of an *equals* method. This method is provided to allow value comparisons between objects.

boolean equals (Object elem);

All classes that override this method must agree with the general contract which specifies that it implements an equivalent relation: it is reflexive, symmetric, transitive, and consistent [7].

Using **Object**, we can send the message *equals to* compare two objects for equality but the other comparison needed in our example, *less-or-equal* and *greater-or-equal*, are not present in Object and can't be added to it.

4.2. Genericity with a simple bound.

We define an interface that declares the methods we required in our example. The interface is called Orderable. A typical code of this interface is shown next.

```
interface Orderable {
    boolean eq (Orderable other);
    boolean le (Orderable other);
    ...
}
```

We declare the class OrderedList with a type parameter T. T is bounded to Orderable. This means that the classes used to create instances of the parameterized type OrderedList must implement the interface Orderable.

```
class OrderedList<T implements Orderable> {
   T listElem;
   ...
   void insert(T elem) { ... if (listElem.le(elem)) ... }
   boolean member (T elem) {
         ... if (listElem.eq(elem)) ... }
}
```

4.2.1. Defining a client class.

We now define a *Point* class. We will use the class *Point* to create an instance of the parameterized class *OrderedList*. According with the constraint defined in the type parameter of *OrderedList*, our class Point must implements the *Orderable* interface.

```
class Point implements Orderable {
   int x,y;
   boolean eq (Orderable other) {
      // we need to cast other to Point before using it
      // if other is an instance of Point compare x and y
      // what to do if other is a subtype of Point?
      // what to do if other is not an instance of Point?
   }
}
```

In class Point the implementation of the method *eq* must have the same signature as its declaration in interface Orderable. The parameter cannot be covariantly specialized because the Java's type system doesn't allow specialization in order to preserve type safety. In this case, the programmer has the responsibility to ensure that the type of the parameter received is the one that is expected or to raise an exception otherwise.

4.2.2. Deriving a class.

We define another class ColorPoint derived from Point. This new class extends the Point class and overrides all the methods defined in Orderable, to manipulate ColorPoint objects. The signature of those methods can not be modified. ColorPoint is a subtype of Point.

```
class ColorPoint extends Point{
   String c;
   ...
   boolean eq (Orderable other) {
   // we need to cast other to ColorPoint before using it
   // compare x, y and c variables
   // what to do if other is not an instance of ColorPoint?
   }
}
```

It is not easy to implement correctly the methods of the Orderable interface. The type parameters of the methods defined in Orderable are of type Orderable. Each class that implements Orderable must take care of the type of the object received as argument at execution time because some problems that are not detected at compile-time can occur. An example is illustrated as follow. Suppose that there is an object *rectangle* created from a class called Rectangle, which is not derived from Point, and implements the interface Orderable. There is also a *point* object created from Point. It could be possible to use the object *rectangle* and compare it with *point* because they both have the *eq* method which receives an Orderable type as argument.

```
point.eq(rectangle);   // compile-time ok!!
```

In this case the method *eq* defined in class Point is going to be executed because the type of the object that receives the message is Point. The type of the argument (Rectangle) is not considered in the selection of the method to be executed because Java has single dispatch.

At compile time the comparison of a point with a rectangle is type-safe because they both are type Orderable but it is meaningless. We need a mechanism to ensure that the type of the object that receives the message and its parameter are of the same type. But this is not possible to express with a simple bound. This situation can be solved using a generalized form of parametric polymorphism which we present in section 4.3.

4.2.3. Using the parameterized type.

We can create instances of OrderedList using Point and ColorPoint classes.

```
OrderedList<Point> pointList = new OrderedList<Point>;
OrderedList<ColorPoint> colorList =
                          new OrderedList<ColorPoint>;
   ...
   pointList.insert(new Point(0,0));
```

In this case, the elements of the ordered list *pointList* are not restricted to be only of type Point but of any type derived from Point. It is possible to insert a ColorPoint object into an ordered list of Point elements because ColorPoint is a subtype of Point and according to subtyping rule; it is safe to use a subtype where a supertype is expected. In this case an object of type Point is expected but we can use an object of type ColorPoint.

```
pointList.insert(new ColorPoint(1,1,"red"));
```

In our example pointList contains two elements, a point and a colored point as illustrated in figure 1.

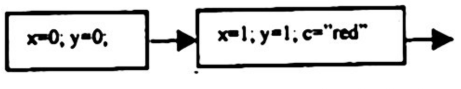

Figure 1. Actual state of pointList.

It is also possible to use as argument a ColorPoint object when the *member* message is sent. This is illustrated in the following example.

```
pointList.member(new ColorPoint(0,0,"white"));
```

When the *member* method of OrderedList class executed, it calls the *eq* method to compare the objects the list with the object received as argument. The

method executed is the one defined in class Point, which will compare only the *x* and *y* variables to decide if the objects are the same. In this case, we will receive a TRUE value that means that ColorPoint(0,0,"white") is in pointList although it is not.
To avoid this, we must define a contract to our implementation of Orderable interface where the *equal* method must implement an equivalence relation.

4.3. Genericity with a recursive bound.

Generic Java relies on F-Bounded polymorphism to allow the definition of parameterized types.
In OrderedList<T implements Orderable<T>> the type parameter T is constrained to implement an interface that is parameterized with itself. A type parameter is needed to define an instance of OrderedList class. The type that can be used to instantiate OrderedList is restricted to implement the parameterized interface Orderable.
When a class is defined with a recursive bound the possibility of reuse its code through inheritance is lost. We illustrate this in the next example.

4.3.1. Interface declaration.

We define a parameterized interface with some abstract methods for any type T.

```
    interface Orderable<T> {
        boolean eq (T other);
        ...
    }
```

4.3.2. Class declaration.

We define a parameterized class called OrderedList, with a type parameter T which is bounded to interface Orderable<T>.

```
    class OrderedList<T implements Orderable<T>> {
            ...
    }
```

4.3.3. Defining a client class.

Now we define a class called Point that provides implementation for all the abstract methods in the parameterized interface Orderable. This class can be derived from other classes that do not implement the same parameterized interface.

```
    class Point implements Orderable<Point> {
            ...
            boolean eq (Point other) {... }
    // the type of the parameter is specialized to type Point
            ...
    }
```

Class Point can be used as an actual type parameter to create instances of OrderedList because Point implements Orderable<Point>. We instantiate the class as follows:

```
    OrderedList<Point> lp = new OrderdedList<Point>;
```

4.3.4. Creating a subclass.

It is possible to define new classes from existing ones. We derive a class called CP from class Point.

```
    class CP extends Point {...}
```

Class CP inherits all from class Point including the methods' implementation of Orderable but those methods are implemented for parameters of type Point and can not be covariantly changed to type CP. Overriding a method does not allow to change the type of the parameters in a covariant way (from class to subclass). The type obtained with the definition of class CP is a subtype of type P.

4.3.5. The problem.

We can not use class CP, or any other derived class form Point, to create instances of OrderedList.

```
    OrderedList<CP> lcp = new OrderdedList<CP>;
        // compile-time error!!
```

This declaration will generate an error at compile time because class CP does not implement the interface Orderable<CP>. It inherited Orderable<Point>.
On the other hand, we are not allowed to define a class that is at the same time a subtype of two interface types that are parameterizations of the same interface. Thus the next definition will also generate an error at compile time.

```
    class CP extends Point implements Orderable<CP> {
        ... }    // compile-time error!!
```

This error is generated due to a restriction imposed by the technique used in the implementation of parameterized types in Java. The explanation is found in [1].

"To support translation by type erasure, we impose the restriction that a class or type variable may not at the same time be a subtype of two interface types which are different parameterizations of the same interface. Hence, every superclass and implemented interface of a parameterized type or type variable can be augmented by parameterization to exactly one supertype."

5. ALTERNATIVES

5.1. Avoid inheritance.

In order to use the class CP to create instances of OrderedList, we need to define it independently of class Point. Here is a third approach to declaring CP:

```
    class CP implements Orderable<CP> {
        ...    // repeat all code from class Point
        boolean eq (CP other) {...}
    }
```

As we notice with this declaration of class CP, code reuse through inheritance is not possible and we have to repeat the code we were reusing from class Point in class CP. Class CP is no longer a subclass of class P and therefore objects of type CP are not subtypes of those with type Point. Nominal subtyping is used in Java and deriving a class from other class yields also to a subtype of the parent class. This is not true in Objective ML, were a class that has binary methods can be extended by inheritance to create new classes but the derived class is not a subtype of the parent class. Binary methods are correctly handled because the type of self is kept open while typing classes [8].

5.2. Inherit first then impose bounds.

We need to choose another option to be able to create a subclass that inherits from class Point and can be used to create instances of the parameterized class OrderedList. A possible solution is to use a kind of clone class to implement the Orderable interface independently of the class hierarchy. Suppose we define the following classes and interfaces.

```
    class Point {...}
    class CP extends Point {...}
        // class CP inherits from class Point
    interface Orderable<T> {...}
    class OrderedList<T implements Orderable<T>> {...}
```

Then we can create subclasses of Point and CP that implement Orderable for them as follows.

```
    class OP extends Point implements Orderable<OP>{
      boolean eq (OP other) { ... }
      ...
    }

    class OCP extends CP
               implements Orderable<OCP>>{
      boolean eq (OCP other) { ... }
      ...
    }
```

The hierarchy created by all the above definitions is shown in figure 2.

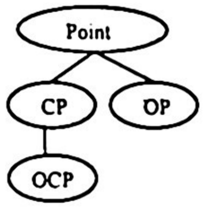

Figure 2. Type hierarchy of classes Point and CP.

Although classes OP and OCP are derived from class Point, the types of classes OP and OCP are not related at all. Classes P and CP can not be used to create instances of OrderedList because they do not implement the interface Orderable. Classes OP and OCP can be used to instantiate OrderedList as follows.

```
    OrderedList<OP> lp = new OrderdedList<OP>;
    OrderedList<OCP> lcp = new OrderdedList<OCP>;
```

The elements that will be part of *lp* will be only of type OP. OP does not have subtypes so all the elements in the collection will be homogeneous. It is the same case for *lcp* all its elements will be of type OCP.

5.3. Multi-methods for Java.

In languages that support multi-methods the selection of the method to be executed at run-time is based in all the parameters not just the receiver. Boyland and Castagna [9] proposed an extension to the Java programming language that support implementation of multi-methods. In their proposal they argue that their implementation of multi-methods for Java can be translated at source level into programs that do not have them. The implementation of multi-methods is modular, type-safe, and allows separate compilation.

6. CONCLUSIONS

One of the most important changes of Java that will be delivered by the end of this year in Tiger, Java version 1.5 compiler, is the introduction of generic types. This extension affects only the source language syntax by adding parameterized types and variables among some other features like automatic boxing of primitive types and enumeration types. No changes to the JVM are made.

In this paper we presented an example and some implementations that require the use of genericity, inheritance and binary methods as well as some problems that can arise when we want to reuse code from a superclass.

The inclusion of generics in Java enhances its expressivity and produces safer code because more errors are caught at compile-time. On the other hand, subclasses derived form recursively parameterized classes, cannot be used to instantiate parameterized types; an alternative solution must be implemented in order to be able to reuse code.

The two models of object-oriented languages [10] [11] seem to be not able (enough) to capture all the features required by a language that support the complexity of object-orientation.

REFERENCES

[1] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Max, Martin Odersky, Sven-Eric Paintz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java^{TM} Programming Language: Participant Draft Specification. April 27, 2001. [Online]. http://java.sun.com

[2] Gilad Bracha, Norman Cohen, Christian Kemper, Martin Odersky, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java^{TM} Programming Language: Public Draft Specification, Version 2.0. June 23, 2003. [Online]. http://java.sun.com

[3] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. Theory and Practice of Object Systems, 1(3):221-242, 1996.

[4] Gilad Bracha, David Stoutamire, Martin Odersky, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA'98*, October 1998.

[5] Luca Cardelli and Peter Wegner. On understanding Types, Data abstraction, and Polymorphism. *Computing Surveys*, Vol. 17 n. 4, pp 471-522, December 1985

[6] Peter S. Canning, William Cook, Walter L. Hill, John Mitchell, and Walter Olthoof. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273-280. ACM, 1989.

[7] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification*. Java series. Addison-Wesley, Reading, Massachusetts, second edition, 2000.

[8] Didier Remy and Jerome Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*. 1998.

[9] John Boyland and Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. *Proceedings of ACM OOPSLA 97 Conference*. Atlanta, GA, USA 1997.

[10] Luca Cardelli, A Semantics of Multiple Inheritance. *Inf. Computation* 76, 138-164.

[11] Giuseppe Castagna, G Ghelli, and G Longo. A calculus for overloaded functions with subtyping. Information and Computation 117,1,115-135 1995. A preliminary version has been presented at the *1992 ACM Conference on LISP and Functional Programming* (San Francisco, June 1992)