Generics in Java and C++ - A Comparative Model

Debasish Ghosh
(dghosh@acm.org)

Anshin Software Pvt. Ltd.
Infinity Building, Tower 1,
Salt Lake Electronic Complex, Sector V, Salt Lake,
Kolkata, India.

1. Introduction

All object oriented programming languages offer various degrees of support for generic programming. C++ offers parametric polymorphism through templates [12], Eiffel [11] offers parameterized classes, Haskell and ML also offers some form of parametric polymorphism. Till JDK 1.5 (beta), Java had no direct support for parameterized types – the user had to depend on the "generic idiom" [1], where variable types were replaced by their typed upper bounds (typically `Object`) to provide the façade of genericity, while introducing casts in referring to the specific uses of these types. The collection classes introduced in JDK 1.2, the Observer pattern implementation of JDK 1.1, all provided enough emphasis on the aspects of fostering genericity in the programming model, although direct language support was missing.

Come JDK 1.5 – we have the Java Generics, supposed to make type-safe container based programming a reality to the programmers. This paper tries to analyze the implementation of the Java Generics and its comparison with the templates of C++. In C++, templates provide one of the vehicles of generic programming – class templates provide support for type-safe generic containers, while standalone function templates represent generic algorithms. Hence templates in C++ are not strictly tied to the object oriented model – in fact, C++ templates go a long way in enriching the multi-paradigm design philosophy.

2. Models of Comparison

The paper looks at the implementation of Generics in Java and C++ from the following models of comparison:

- Translation model, which discusses the differences between the two languages in the way in which generic code is translated downstream
- Type model, which discusses the differences in the implementations of type inference, type aliasing and associated types in C++ templates and Java Generics.
- Security model, which highlights the security loopholes, if any, in any of the implementations of Generics

The paper concludes with the restrictions imposed on the programming model by the implementation of the Generics in Java.

3. Translation Model

One of the major driving forces behind the design philosophy of Java Generics was to provide backward compatibility of legacy Java code, which exploited the "generic idiom" [1] using the extensive library of Java collection classes. In order to retrofit the legacy library code to the generic variant, the generic Java implementation has adopted the same representation policy for the raw type and the parametric type. The parametric type `Collection<T>` can be passed safely where the corresponding raw type `Collection` is expected. Hence the translation model of Java Generics is based on the "type erasure" model [1], where the translation erases the generic type parameter, replaces them by the bounding type (`Object`) and inserts casts to the actual places of usage. In order for method overriding to work correctly, the translation process also adds "bridge methods", as illustrated in the following example [Figure 1]. In the figure below, the bridge method needs to be introduced since the erasure of the argument and return type in the subtype D.id() differs from the erasure of the corresponding types in type C.id(). Because of this model, Java loses all run-time type information, thereby eliminating those operations which require run-time types e.g. instanceof operation, casts, explicit instantiation of the naked generic type etc. A more detailed review of the restrictions in Generic Java is described in Section 6.

```
// the example is taken from [10]
class C<A> { abstract A id(A x); }
class D extends C<String> { String id(String x) { return x; } }
```

This will be translated to:

```
class C { abstract Object id(Object x); }
class D extends C {
    String id(String x) { return x; }
    // bridge method introduced by the translator to make
    // overriding work correctly
    Object id(Object x) { return id((String)x); }
}
```

Figure 1: Translation by Type Erasure in Java

The translation model of C++ is based on instantiation – no static type checking is done on the generic code. The compiler generates separate copies of the component (class or function) for each instantiation with a distinct type and the type checking is performed after instantiation at each call site. Type checking of the bound types can only succeed when the input types have satisfied the type requirements of the function template body. It is because of this, that misleading error messages are thrown in by the compiler when a generic component is instantiated with an improper type.

Thus the C++ model is loss-less in the sense that no type information is lost during runtime, though the implementation of the translation suffers from the drawbacks of the possibility of introducing code bloats and misleading error messages. On the other hand, the Java model emphasizes on retrofitting the raw types with the generic counterparts, but the type-erasure model delegates the generic types to "second class" status compared to the conventional types.

4. Type Model

In Java Generics, type requirements can be defined on arguments as a set of formal abstractions – this feature is called constrained genericity. The generic types of the classes have to honor these requirements in order to participate in the valid instantiation. This constraint on the generic type is known as *"concept"*, following the terminology of Stepanov and Austern. Java Generics use interfaces to represent a concept and employ the mechanism of subtyping to *model* a concept – any type T modeling a concept C will have to implement the corresponding interface. The following example (Figure 2), due to [2], gives the Java representation of three graph concepts and an adjacency list data structure modeling these concepts. The `implements` clause makes the `adjacency_list` a model of Vertex List Graph and Incidence Graph concepts.

```
public interface VertexListGraph<Vertex,
                      VertexIterator extends Iterator<Vertex>> {
    VertexIterator vertices();
    int num_vertices();
}

public interface IncidenceGraph<Vertex, Edge,
                      OutEdgeIterator extends Iterator<Edge>> {
    OutEdgeIterator out_edges(Vertex v);
    int out_degree(Vertex v);
}

public interface VertexListAndIncidenceGraph<…>
        extends VertexListGraph<…>, IncidenceGraph<…> {}

public class adjacency_list
        implements VertexListAndIncidenceGraph<Integer,
                      Simple_edge<Integer>, Iterator<Integer>,
                      Integer, Iterator<simple_edge<Integer>>,
                      Integer, Iterator<simple_edge<Integer>>> {
……
}
```

Figure 2: Subtyping models a concept in Java

Modeling constrained genericity through subtyping has the problem of increased coupling between the generic type definition and the constraints on the generic parameters. Subtyping relationship is statically defined during class definition – hence existing types cannot be made to model new concepts without changing their definitions. C++ does not offer any means to constrain the template parameters. [6] describes the logic to keep this feature off the language – in spite of repeated discussions in many C++ Standardization meetings, members could not agree upon a suitable mechanism of enforcing the constraints which would cater to the varieties of containers and generic types used in practice. However, techniques for checking constraints in C++ have been implemented as a library using the notion of compile time assertions [8].

Java Generics come without two of the very important features of generic programming – type aliasing and associated types. C++ templates offer both of them, type aliasing through `typedef` s and the associated types through the traits mechanism [7]. Though `typedef` s are not part of the C++ templates *per se*, but without them writing generic code often becomes extremely verbose and error prone. The example in Figure 3 (adopted from [3]) illustrates this verbosity. The instantiation of the class `dijkstra_visitor` could have been much more elegant had Java given the support of declaring type aliases.

```
dijkstra_visitor<VertexListAndIncidenceGraph<Vertex,
                                             Edge,
                                             VertexIterator,
                                             OutEdgeIterator>,
                mutable_queue<Vertex, indirect_cmp<Vertex,
                         Distance, DistanceMap, DistanceCompare>>,
                WeightMap,
                PredecessorMap,
                DistanceMap,
                DistanceCombine,
                DistanceCompare,
                Vertex,
                Edge,
                Distance> vis
      = new dijkstra_visitor<VertexListAndIncidenceGraph<Vertex,
                                                         Edge,
                                                         VertexIterator,
                                                         OutEdgeIterator>,
                             mutable_queue<Vertex,
                                 indirect_cmp<Vertex, Distance,
                                      DistanceMap, DistanceCompare>>,
                             WeightMap,
                             PredecessorMap,
                             DistanceMap,
                             DistanceCombine,
                             DistanceCompare,
                             Vertex,
                             Edge,
                             Distance> (Q, weight, predecessor,
distance, combine, compare, zero);
```

Figure 3: Java has no type aliasing

Traits offers a mechanism to use encapsulated types – Java classes can only encapsulate data members and methods, but not types. Hence in Java associated types of a concept become extra type parameters, resulting in increased verbosity and repetition of type constraints. In Figure 4 (adopted from [3] and [4]), the prototype for `breadth_first_search()` in C++ contains much lesser number of parameters in the type parameter list, since the type of the second parameter, which is the vertex type, is associated with the graph type. In the corresponding Java prototype, the generic interface has to be explicitly parameterized by specifying all the type parameters (and the associated ones as well), resulting in more verbosity (see example below).

```
// breadth_first_search prototype in C++
template <class VertexListGraph, class Buffer, class BFSVisitor,
          class ColorMap>
  void breadth_first_search
    (const VertexListGraph& g,
     typename graph_traits<VertexListGraph>::vertex_descriptor s,
     Buffer& Q, BFSVisitor vis, ColorMap color)

// breadth_first_search prototype in Java
public static <Vertex,
               Edge extends GraphEdge<Vertex>,
               VertexIterator extends java.util.Iterator<Vertex>,
               OutEdgeIterator extends java.util.Iterator<Edge>,
```

```
                Vis extends Visitor,
                ColorMap extends
                     ReadWritePropertyMap<Vertex,java.lang.Integer>,
                QueueType extends Buffer<Vertex>>
  void breadth_first_search(VertexListAndIncidenceGraph<Vertex,
                                Edge,VertexIterator,OutEdgeIterator> g,
                                Vertex s, Visitor vis, ColorMap color)
```

Figure 4: Java has no support for associated types

5. Security Model

Java Generics translation model replaces all generic types by the bound type, `Object`, thereby removing all type information from the runtime system. This is referred to as the homogeneous model of implementing Generics, since all generic types are mapped to a single supertype. C++ implements the heterogeneous model, where the compiler generates a separate copy of the type for each specific instantiation of the type parameter. Hence a generic container Stack<T> will generate separate concrete classes Stack<int> for integers, Stack<double> for double precision numbers, Stack<string> for string data type. While in Java, Stack<T> will be instantiated as Stack collection class.

As mentioned in [5], the homogeneous model of implementing Generics opens a potential security hole in the type system during run-time, since the translation erases all type information. Consider the example from [5]:

```
Class BroadcastList<C extends Channel> {
    C channels[];
    void add(C c) { … }
    void broadcast(String s) {
        int I;
        for(I = 0; I < channels.length; i++)
            channels[i].send(s);
    }
    ……
}

Class EncryptedChannel extends Channel;
Class UnencryptedChannel extends Channel;

BroadcastList<EncryptedChannel> list = ……;
list.add(new EncryptedChannel());
(*)
list.broadcast("Hello");
```

Figure 5: Exploiting the Security Loophole

Because of the homogeneous implementation, during instantiation of the generic class `BroadCastList`, `C` will not contain the exact type, since the translation of Java Generics will erase C. Hence it becomes vulnerable to a malicious program who can add an unencrypted channel to the list (possibly at position marked by (*)) through hand-written byte codes, which will bypass all compile time checking and the compiler generated run-time checks guaranteeing

type safety. JDK 1.5 (beta) includes a workaround to prevent this security hole; it has a means of constructing a dynamically typesafe view of a specified collection (`Collections.checkedCollection()`). This ensures that any attempt to insert an element of incorrect type will result in a `ClassCastException`. But this approach introduces a counter-intuitive programming model specifically to address an existing loophole in the security of the JVM.

In C++, this security loophole does not exist since strict type-checking is employed at the points of instantiation of templates. However, this heterogeneous translation model employed by C++ will not fit with the current security model of the Java virtual machine. The JVM offers two levels of visibility – global level (public) and package level. Hence in some cases it becomes impossible to find out the proper package where the instantiated class will be placed honoring the visibility model of the JVM. For details, the reader is referred to [1].

6. Restrictions in Java Generics

One of the major driving forces behind the design of the Java Generics has been maintenance of backward compatibility with the existing language semantics and implementation. The user should be able to retrofit the existing library classes with the generic components without any change in code. The type-erasure model nicely fits in this paradigm and provides the ideal way to introduce the expressiveness and type-safety to the Java programming language. But the loss of runtime type information and the homogeneous translation model have introduced some serious restrictions to the programmer. Some of the major ones are mentioned below. For details on these restrictions, the reader is referred to [5].

a) Primitive types cannot be type parameters in a generic component (method or class). Class instantiations like `Vector<int>, LinkedList<String>` are not allowed in Java Generics, since the primitive parameters do not have a bound type (`Object`).

b) Type variables (belonging to generic type parameters) cannot appear in the `catch` clauses of Java exceptions, though they are allowed in `throws` lists.

c) Casting to a generic type is not allowed, unless the target of the cast can be statically deduced by the type checker. Since no run-time type information is maintained, the cast can succeed only if the compiler can implement the cast of a type `C<T>` by its type erasure (`C`). Because of this same reason, the following programming practices are restricted in Generic Java:
   - Using interfaces like `Cloneable()`, which require casting of the object returned by `clone()` to the desired type cannot be done unless the parametric type information can be deduced statically by the type checker.
   - Using instanceof operations, which also require run-time type information.
   - Using dynamic allocation of the naked type, as in new T[]

7. Conclusion

Generics in Java has been designed to increase the expressiveness and type-safety of generic programming. But the lack of run-time type information in the language has forced lots of

restrictions in the programming model. But even then, the retrofitting of the existing Java libraries with the current implementation of the Generics has ensured complete backward compatibility. C++, because of its heterogeneous translation model, does not suffer from these restrictions imposed by the type-erasure idiom. But creation of a separate copy of each class for each instance of the generic parameters makes the compilation process slow and has the potential of introducing code bloats and generating misleading error messages. Even then, C++ templates is a Turing complete language with much more expressive power of generic programming. Additional features like associated types and type aliasing make generic programming an enjoyable experience in C++. Comparatively, Java Generics may seem to be a syntactic sugar that promotes casting from the programmers' level to the bytecode level. The Java compiler translates generic Java code to non-generic Java bytecode, for reasons of compatibility with previous versions of Java, but losing important run-time type information. There have been some suggestions of alternative implementations [13], where run-time type information can be preserved and used effectively at the expense of a little additional overhead on part of the JVM.

8. References

[1] Gilad Bracha, Martin Odersky, David Stoutamire, Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA)*, 1998.

[2] Ronald Garcia, Jaako Jarvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA)*, 2003.

[3] Implementation of [2] (http://www.osl.iu.edu/research/comparing)

[4] The Boost Graph Library (http://www.boost.org/libs/graph)

[5] E. Allen, R. Cartwright. The Case of Runtime Types in Generic Java. In *Principles and Practice of Programming in Java*, 2002.

[6] B. Stroustrup. *Design and Evolution of C++*. Addison-Wesley, 1994.

[7] N. C Myers. Traits: a new and useful template technique. In *C++ Report*, June 1995

[8] J. Siek, A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[9] Ole Agesen, Stephen Freund, John C. Mitchell. Adding parameterized types to Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA)*, 1997.

[10] G. Bracha, N. Cohen, C. Kemper, S. Marx, et al. JSR 14: Add Generic Types to the Java Programming Language, April 2001. http://www.jcp.org/en/jsr/detail?id=014.

[11] B. Meyer. *Eiffel: the language*. Prentice Hall, New York, NY, first edition, 1992.

[12] David Vandevoorde, Nicolai M. Josuttis. *C++ Templates: The Complete Guide*, Addison-Wesley, 2002.

[13] R. Cartwright, G. Steele. Compatible Genericity with run-time types for the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA)*, 1998.