A Comparative Analysis of Generic Programming Paradigms in C++, Java and C#

Arijit Khan and Shatrugna Sadhu
Department of Computer Science, University of California, Santa Barbara
{arijitkhan, ssadhu}@cs.ucsb.edu

Abstract: Generic programming is programming without named types, which allows us to effectively avoid the static typing requirements for statically typed programming languages. It is an increasingly popular and important paradigm for software development and many modern programming languages provide basic support for it. In this report, we make a comprehensive survey of generic programming in three of the most widely used languages, e.g. C++, Java, and C#.

I. INTRODUCTION

"Generic programming is 'programming with concepts,' where a concept is defined as a family of abstractions that are all related by a common set of requirements" – David R Musser. The basic objective of Generic Programming is code reusability, i.e. use the same module for different types. It allows some code to effectively avoid the static typing requirements for statically typed programming languages [3]. Therefore, it can rightly be called as the type-safe polymorphism [2]. Prior to the existence of the Generics feature set, this purpose was solved by specifying the type of the objects within the data structure as Object, and then casting to specific types at runtime. However, this technique had several drawbacks including lack of type safety, poor performance and code bloat [5].

Note that, code reusability is different from class reusability or inheritance [4], where we reuse the members of a class, but types remain intact. Reusable modules and classes reduce implementation time, increase the possibility that prior testing and uses have eliminated bugs, and localizes code modifications when a change in implementation is required. Generic programming is an increasingly popular and important paradigm for reusable libraries and software development, and therefore, many modern programming languages provide basic support for it. As the generic programming paradigm gains momentum, it is important to clearly and deeply understand the language issues. This paper presents a comparative analysis of generic programming paradigms in 3 different programming languages namely C++, Java and C#.

II. Templates in C++

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. Generic type can be used for any type or non-type constant. While implementing class template member functions, the definitions are prefixed by the keyword template. Here is the example of a 'class template' xyz.

```
template <class itemTye>
class xyz {
   private:
      itemType a, b;
   public:
     xyz (itemType p, itemType q) {
        a = p;
        b = q;
     }
} ;
```

The followings are two sample driver classes (also known as 'template class' or 'generated class') that use the class template xyz.

```
int main() {
   xyz <float> floatobject (3.6, 8.2);
   xyz <int> intobject (3, 8);
}
```

In the main(), we instantiated an xyz of float and another xyz of int. So, during compile time, the compiler instantiates (generates) two distinct class types and gives its own internal name to each type. When the compiler instantiates a template, it literally substitutes the actual parameter for the formal parameter throughout the class template, which is similar to perform a search-and-replace operation in some text editor. This process is commonly known as "*Instantiating a class*" [6]. So, there is no run-time overhead, but compilation is slower. For example, when the compiler encounters xyz <float>, it generates a new class by substituting float for every occurrence of itemType in the class template. The result is the same as if we have written the following:

```
class xyz_float {
  private:
    float a, b;
  public:
    xyz(float p, float q) {
      a = p;
      b = q;
    }
};
```

Once the template class is instantiated, one can instantiate objects of that type. For example, floatobject is an object of xyz of float (xyz_float), and intobject is an object of xyz of int (xyz_int).

```
xyz_float floatobject (3.6, 8.2);
xyz_int intobject (3, 8);
```

Note that the instantiations of templates are somewhat similar like macro expansions in C. Although macros and templates are two separate concepts, and their differences are explained in [1].

One interesting fact about template is that, if Foo is a subtype (subclass) of Bar, then xyz<Foo> is not a subtype (subclass) of xyz<Bar>; where xyz is a generic class. This is because of the fact that xyz<Foo> and xyz<Bar> have their own internal class representations, say xyz_Foo and xyz_Bar respectively, as explained above. Now although the name of the data-members and member-functions are same in both these classes, the types of data-members and the signatures of member-functions are different (In xyz_Bar, they are of Bar types, whereas in xyz_Foo, they are of Foo types). Therefore, xyz<Foo> can not a subtype of xyz<Bar>. This is, in fact, a common property of Generics, and in the following sections, we show how Java and c# implements this property, although in a different manner.

Templates are not normal functions or classes. They are compiled on demand, i.e., the code of a template is not compiled until an implicit or explicit instantiation with specific template arguments is required [7]. At that moment, when an instantiation is required, the compiler generates a function or class specifically for those arguments from the template. For example, consider the following code:

```
template <class T>
class xyz {
  public:
   xyz() {…} ;
   ~xyz() {…} ;
   void f(){…} ;
   void g(){…} ;

} ;

int main() {
 xyz<int> zi ;         // implicit
                       // instantiation
                       // generates class xyz<int>
 zi.f() ;             // generates
                      //  function
                      //xyz<int>::f()

xyz<float> zf ;     //implicit instantiation
                     // generates class xyz<float>
 zf.g() ;            //and generates function
                       // xyz<float>::g()
  return 0;
}
```

This time in addition to the generating classes xyz<int> and xyz<float>, with constructors and destructors, the compiler also generates definitions for xyz<int>::f() and xyz<float>::g(). The compiler does not generate definitions for functions, nonvirtual member functions, class or member class that does not require instantiation. In this example, the compiler did not generate any definitions for xyz<int>::g() and xyz<float>::f(), since they were not required.

Now consider the following instantiation of generic class xyz.

```
int main() {
  xyz<int>* p_zi ;      //instantiation of class
                         // Z<int>  not required
  Z<float>* p_zf ;   //instantiation of class
                         // Z<float> not required
  return 0 ;
}
```

This time the compiler does not generate any definitions, since there is no need for any definitions. It is similar to declaring a pointer to an undefined class.

In C++, the code that defines a class can be split into two files: the first part (also known as declaration) specifies the member functions (methods) and data members (fields) of the class. This code goes into a header file (a file with the extension .h). The definitions (or implementations) of the member functions of that class are placed in some other file with same name, but having the extension .cpp. The reason for splitting up the code is that it is generally a good idea to separate the interface from the implementation. Someone who wants to use a class may only need to know what operations are available; it may not be necessary to know all the details about how the operations are actually implemented. Besides, it helps to design modularity, since whenever the function definition is changed, we need to recompile only the .cpp file containing that function definition; but not the main program using the function. See the following example:

```
// ORIGINAL version of xyz.h
class xyz {
  private:
     int a, b;
  public:
     xyz(int x, int y);
};
```

```
// ORIGINAL version of xyz.cpp
#include "xyz.h"
xyz::xyz(int x, int y) {
  a = x;
  b = y;
}

// main.cpp
#include "xyz.h"
int main () {
  xyz object (3,6);
  return 0;
}
```

However, templates are compiled when required, and this forces a restriction for class template: the implementation (definition) of a class template or function template must be in the same file as its declaration. That means that we cannot separate the interface in a separate header file, and that we must include both interface and implementation in any file that uses the templates. Consider the following example:

```
// ORIGINAL version of xyz.h
template <class T>
class xyz
{
 private:
   T a, b;
 public:
   xyz(T x, T y);
};

// ORIGINAL version of xyz.cpp
#include "xyz.h"
template <class T>
xyz<T>::xyz(T x, T y) {
  a = x;
  b = y;
}

// main.cpp
#include "xyz.h"
int main() {
  xyz <int> intObject (3, 6);
  return 0;
}
```

Now, if one compiles both xyz.cpp and main.cpp as before, error due to unresolved externals will be reported. This is because there is no use of the constructor within xyz.cpp, therefore, there is no instantiations that needs to occur from there. So, the constructor is not generated from the compilation of xyz.cpp. Now, when main.cpp tries to call that constructor during its own compilation, error due to unresolved externals is displayed. Since the complete implementations of the source code reside in the header file, the users need to recompile the main program that uses the template, whenever the implementation changes. Thus, templates have some drawbacks in the area of modularity.

Each template class or function generated from a template has its own copies of any static variables or members. Moreover, Static variables can be of Generic types in template. Each instantiation of a class template has its own copy of any static variables.

```
template <class T>
class X {
   public:
         static T s ;
} ;
template <> int X<int>::s = 3 ;
template <> char* X<char*>::s = "Hello" ;

int main() {
      X <int> intObject1 ;
      X <char*> stringObject ;
      X <int> intObject2;
}
```

Here X <int> has a static data member s of type int and X <char*> has a static data member s of type char*. Clearly, the same static data member s of type int will be used by both intObject1 and intObject2.

Another drawback of C++ template is that the arguments are not type checked — but instead the body of the function template is type checked. So, when a function template is misused, the error message points to lines within the template. Therefore, the template implementation is unnecessarily exposed to the user and the actual reason for the error becomes harder to find.

Nevertheless, C**++** templates still provide type safety with genericity; and unlike Java Generics (which will be discussed in the following section), no downcast mechanism is needed when constructing generics. Of course, C**++** itself is not fully type safe because of various loopholes that exist in the type system. These loopholes, however, are orthogonal to templates. The template system does not introduce new issues with respect to type safety [2].

III. Java Generics

Java's generic programming features are based on Pizza project's GJ (Generic Java) compiler [8]. Classes, Interfaces and methods can be genericized in Java. Unlike templates, the actual types (which will eventually substitute the generic types) can only be of reference types, and primitive types. This is explained later. The following is a small code for the definitions of the interface Iterator in package java.util:

```
public interface Iterator<E> {
   E next ();
   Boolean hasNext ();
}
```

Here, E is the formal (generic) type parameter of the interface Iterator. Invocation of the generic type declaration Iterator can be done as follow:

```
Iterator <list> iterator = new <list> Iterator ();
list p = iterator.next ();
```

A generic type declaration is compiled once and for all, and turned into a single class file, just like an ordinary class or interface declaration. Compiler removes all of the "genericity", and then uses a standard java compiler to convert the source to bytecode. To translate Generic Java (GJ) into Java, following steps are performed:

a) Delete generic parameters (e.g., <E>); which is called *erasure* in GJ

b) Replace all occurrences of type parameters (e.g., E in Iterator) by Object in the byte code. Note that Object is the supertype of all reference types in Java.

c) Insert appropriate casts where necessary.

Therefore, after compilation, the previous interface definition and invocation become equivalent to as follow:

```
public interface Iterator {
  Object next ();
  Boolean hasNext ();
}

Iterator iterator = new Iterator ();
list p = (list) iterator.next ();
```

Note that the type-cast in the intermediate bytecode, as shown above, is necessary; since iterator.next() returns an Object type, but this is assigned to a variable of list type. However, there is no need to check the correctness of this type-casting at run time, since the compiler itself can assure that this type-casting is correct. When it compiles the following instantiation, it becomes certain that, whatever iterator. next() returns, must be of list type; and therefore, the type-casting in the intermediate code can never fail internally.

```
Iterator<list>iterator=new<list>iterator()
```

However, if the same thing is implemented straightforwardly without generics as below, then the type-casting would indicate only a promise to the compiler that, we shall assign a list to the return type of iterator.next() during run time. So, compiler would allow it, but it is again checked during run time whether we have really assigned some List to the return type of Iterator.next().

```
public interface Iterator {
  Object next ();
  Boolean hasNext ();
}

Iterator iterator = new Iterator ();
list p = (list) iterator.next ();
```

Therefore, GJ enables runtime checks to be moved to compile time. However, the Intermediate Type Casting increases Compile-time Complexity. Java Generics, therefore, is a syntactic sugar to help with the new for each reference types.

The following diagram (Fig. 1) on Java subtype-relation [9] explains why generic types cannot be substituted by primitive types. The smallest common supertype of Object and any primitive type is the undefined type T. So, if we substitute a generic type by a primitive type, and later use that variable, it will fail during Bytecode Verification.

Since a generic type declaration is compiled and turned into a class file, unlike template, only Signature of the class is sufficient to use it in Java (i.e. its declaration in the current file is not necessary). Moreover, once compiled, the JVM does not require source of generic code to be available. This generated bytecode can theoretically run on older JVMs. In fact, GJ was actually designed to maintain backward compatibility with pre-Java5 code.

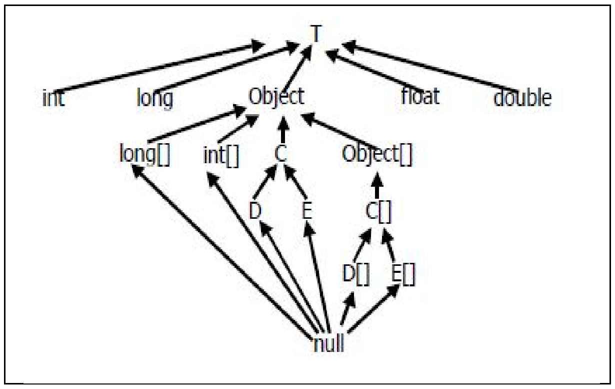

Figure 1: Java Subtype Relation

As mentioned earlier, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is not the case that G<Foo> is a subtype of G<Bar>. In general we cannot do the following [10]:

```
Class Bar {
  int a;
   …..
}
Class Foo extends Bar {
  int b;
   ……
}

class G <T> {
  T item;
   ……
}
G <Foo> f1 = new G <Foo> ();
G <Bar> b1 = f1;  //compile time error
```

As shown in Figure-2, although, object f1 and b1 both have the data member named ‘item’, f1.item is of Foo type, whereas b1.item is Bar type as shown in the previous diagram. So, f1.item and b1.item are basically different (only their names are similar). So b1 = f1 will produce compile time error. It also explains why G<Bar> is not a super type of G<Foo>.

Because there is only one copy of a generic class, static variables are shared among all the instances of the class, regardless of their type parameter. As a result, the type parameter cannot be used in the declaration of static variables or in static methods. Static variables and static methods are "outside" of the scope of the class's parameterized types.

As mentioned earlier, Generic type parameters are erased when compiled; so objects of a generic class with different actual type parameters are the same type at run time. As a result, there is no way to tell at runtime which type parameter is used on an object. Besides, we cannot create an instance of Generic Type parameter at runtime. The following is an example:

```
class xyz<T> {
  T b;
  xyz() {
    b=new T();    //will create an instance of
                  //Generic type parameter at
                  // runtime; Compile error.
  }
}
```

IV. C# Generics

Generics are the most powerful feature of C# 2.0. Although both C# and Java provide a mechanism for creating strongly typed data structures without knowing the specific types at compile time, but they are drastically different in implementation and capabilities [11].

In C#, there is explicit support for generics in the .NET runtime's instruction language (IL). When the generic type is compiled, the generated IL contains place holders for specific types [12]. At runtime, when an initial reference is made to a generic type (e.g. List<int>) the system checks if anyone already asked for the type or not. If the type has been previously requested, then the previously generated specific type is returned. If not, the JIT compiler instantiates a new type by replacing the generic type parameters in the IL with the specific type

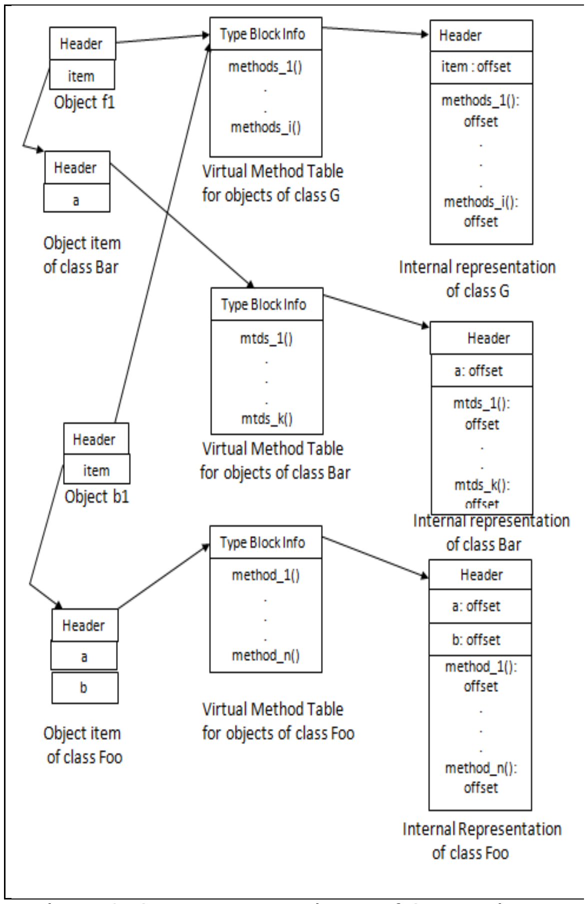

Figure 2: G<Foo> not a subtype of G<Bar> in Java

(e.g. replacing List<T> with List<int>). It should be noted that, requested type, therefore, can be both reference types and primitive types. If the requested type is a reference type, then the generic type parameter is replaced with Object. However unlike java, there is no casting done internally by the .NET runtime when accessing the type. So, it reduces compile-time type-casting complexity, which was a major drawback of java generics.

Since type parameters are not erased during compile time, .NET supports instantiation of generic type at run time. C# also has extensive support for querying generic information at runtime via System.Reflection, e.g. System.Type. GetGenericArguments().

However, there are still some translations performed by the compiler. These translations have been standardized for Common Language Subset use, though these specific changes are not required. The *actual* type name for generic types is the original type name, followed by ``' and the number of generic type parameters. Thus List<T> has the IL name List`1. This allows multiple different generic types to share the same name as long as they have a different number of generic type parameters, e.g. one can have the types Foo, Foo<T>, and Foo<T,U> all in the same namespace. This impacts type lookup through reflection: *Type.GetType ("System. Collections.Generic.List")* will fail, while *Type. GetType("System. Collections .Generic.List`1")* works. Note that, since C# 2.0 has intermediate representation specifically for generics, a generic code will not work in previous versions of C#.

Finally, similar to C++ and unlike Java, static members in C# can be of Generic types, and the Generic class with a static member has one static memory location per Generic + type parameter combination [13].

V. CONCLUSION

We performed a comprehensive study of generic programming in three of the most widely used languages, e.g. C++, Java, and C#. In C++ templates, the Generic types can be replaced by reference types, as well as primitive types. However, compilation is slower because of separate instantiation for each type. Moreover, it is not type safe, and also suffers from modularity problem. Generic Java (GJ), on the other hand, is completely a syntactic sugar, and one of the main objectives was to make it compatible with pre-Java5 codes. The type-casting at compile time increases complexity, and because of type-erasure property, the actual types are not known during run time. Moreover, because of the nature of Java subtype relation, actual parameters can only be of reference types, and not primitive types. C# generic is an improvement over both of them, since the compiled IL of generic types contains place holders for specific types. It is only during the run time, when the JIT compiler instantiates a new type by replacing the generic type parameters in the IL with the specific type. So, C# generic compilation is more efficient than that of template and java generics, and also it has a better run time support. Moreover, unlike java, the actual types are known at run time, and therefore, C# supports instantiation of generic types during run time. We also performed a comparison how static members react inside a generic class for these three languages. Finally, we have shown if Foo is some subtype of Bar, why it may not necessary that G<Foo> will be a subtype of G<Bar>.

References:

[1] G. McComb, "WordPerfect 5.1 Macros and Templates", *http://gmccomb.com/ wpdos /intro. htm*

[2] R. Garcia, J. J¨arvi, A. Lumsdaine, J. Siek, and J. Willcock, "A Comparative Study of Language Support for Generic Programming" in Proc. *OOPSLA 2003*, pp. 115-134.

[3] E. Allen, "Diagnosing Java code: The case for static types", *http://www.ibm.com/developer works /java/library/j-diag0625.html.*

[4] R. E. Johnson and B. Foote, "Designing Reusable Classes", Journal *of Object-Oriented Programming* June/July 1988, Vol. 1, No. 2, pp. 22-35

[5] D. Obasanjo, "A Comparison Of Microsoft's C# Programming Language To Sun Microsystems' Java Programming Language", *http://www.25hoursaday.com/CsharpVsJava.ht -ml*

[6] Microsoft Visual C++, "C++ Template Tutorial", *http://www.iis.sinica.edu.tw/ ~kathy/ vcstl/default.htm*

[7] Comeau C++ Template FAQ, "Tech Talk About C++ Templates", *http://www.comeauco mputing.com/ techtalk/templates/*

[8] The Pizza Compiler, Version 1.1 (January3, 2002), *http://pizzacompiler.sourceforge.net/*

[9] X. Leroy, "Java Bytecode Verification: An Overview", in Proc. *Computer Aided Verification*, pp. 265-285, 2001

[10] G. Barcha, "Generics in the Java Programming Language", *http://java.sun.com /j2se/1.5/pdf/generics-tutorial.pdf*

[11] J. Pryor, "Comparing Java and C# Generics", *http://www.jprl.com/Blog/archive/development /2007/Aug-31.html*

[12] J. Lowy, "An Introduction to C# Generics", Visual Studio 2005 Technical Articles, *http://ms dn.microsoft.com/en-us/library/ms379564.aspx*

[13] NoopMan, "Generic Types Don't Share Static Members", *http://www.codeproject.com /KB/cs/GenericStatic.aspx*