Generic Programming in C++ and Java

Shilpa Mathur

Bhagwant University, India

ABSTRACT
This paper is about Generic Programming in Java and C++. One of the main motivations for including generic programming support in both Java and C++ is to provide type-safe homogeneous containers. To improve the support for generic programming in C++, we introduce concepts to express the syntactic and semantic behavior of types and to constrain the type parameters in a C++ template.

KEYWORDS
Generic programming, Constrained Generics, Parametric Polymorphism, C++ templates, C++ concepts, Standard Template Library (STL).

1. TERMINOLOGY

Generic programming can be seen simply as the act of using type parameters. A broader definition was given by the organizers of a seminar on generic programming: Generic programming is a sub-discipline of computer science that deals with finding abstract representation of efficient algorithms, data structures and other software concepts and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- *Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.*
- *Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.*
- *When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.*
- *Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.*

2. GENERIC PROGRAMMING

Generic programming is implemented differently in different programming languages. Generic programming is a kind of polymorphism. Each variable and routine which can have different types depending on certain circumstances during the execution of a program is polymorphic.Genericity is also called parametric polymorphism.

2.1. JAVA

Java is an object oriented language that allows programmers to create generic classes. When creating generic classes the developer puts at least one type parameter in angle brackets after each class or interface declaration. These type parameters may be bound or unbound in java.

2.1.1 Genericity internals

The compilation of a generic class in Java is homogeneous for each instantiation of generic type. The complier removes all type information related to type parameters. This process is called type erasure. By erasing the type parameters, raw types are created. This is done by Java compiler in order to be backward compatible with older non-generic libraries. The raw type of Foo<Integer> is Foo. Furthermore, it is impossible to use the type parameter at run-time. It is removed by the compiler. An attempt to compile the following class results in an error:

```
public class FooBar<T>
{
public static void main(String args[])
{T t = new T();  //not allowed  }
}
```

2.1.2 Wildcards

In Java the type of generic container can be unknown. Instead of a type that replaces the type parameter the wildcard ? is used. We can use this wildcard bound or

Manuscript received September 5, 2011
Manuscript revised September 20, 2011

unbound. When using a wildcard for a generic container we must be aware of the fact that some constraints exist on variables which represent such containers. This code snippet shows the usage of a wildcard:

```
Public static void showAll(List ? extends Printable> elems){
for (Printable p : elems) System.out.Println(p);  }
```

This static method showAll() can be called only with list which contain subtypes of Printable. Our Printable interface declares a toString method. Therefore for each element in this list we can output it to standard out. The complier can statically ensure that each element provides a toString method, because each element implements the Printable interface. If an object in java provides the toString method, then we can output its string retranslation to standard out. If you are familiar with Java you will know that we don't need a Printable interface which declares the toString. Everything (except primitive types) is a subtype of Object and Object already implements the toString method. Our example is used to demonstrate wildcards. Our declaration type of our elems variable can be List<? Extends Object> and this declaration is equivalent to List<?>.
Furthermore we must be aware of the fact that within showAll we can't perform any writing operation which would insert objects into our elems list. If extends is used our container is accessible read only.
We can also use super keyword. This allows only super types of a bound to replace the type parameter. In this example the bound is Item. For example:

```
public static void initList ( List < ?  super Item> items) {
// in the loop generate some 'Items' and insert them into
//items
items.add(. . .);
}
```

Into the items we can add anything of type Item or its super type.in this case we can't read from items. The list items is only writable.

2.1.3 Library Development

In Java the type variable can be used without a bound. Often it is not required to constrain the library user. Consider any unconstrained generic container from the JDK .In this case it is even desirable that a user can create a collection of any type. When no constraint is specified, the Java compiler uses implicitly Object as a constraint. Each reference type (except the primitive types: int,double,..)in Java is a subtype of Object.

If the library developer wants to constrain a generic type, he/she uses the keyword extends. Extends state that only subtypes can be replaced for the type variable. Consider the example:

```
Interface UploadAble {
String getPath() ; }

Class ToUpload<Payload extends UploadAble>{
private List<Payload>elems;
public void uploadAll() {
for(Payload p : elems) {
a.Uploader.upload( p.getPath());
}
}
// 'add' also implement here
```

This generic ToUpload class has only one type parameter. We can use this generic class as a container for objects which are uploadable.An object is uploadable if it implements the UploadAble interface and provides the getpath method. For type parameter Payload only subtypes of UplaodAble can be inserted. Subtypes of UploadAble must implement the getPath method which is called within uploadAll.With this approach the library user is constrained in such a way that the compiler ensures that the type inserted for Payload must implement the getPath method.
In Java it is also possible that the library developer uses type parameters recursively. This is necessary when binary methods are implemented by classes which implement generic interfaces. Consider the code snipped given in Figure 1.
The class Integer is a simplified form of Java's standard class with the same name.Integer is extended from the generic Comparable<A> interface. This may be confusing, but it is well defined. This type of genericity is based on the formal model of F-bounded Polymorphism for OO programming.
The advantage of this approach is that the type of formal parameter is defined by the type parameter not by subtyping. This avoids constraints on subtyping (covariance and binary methods) and it ensures that this and that have the same declared type.

Figure1

```
interface Comparable<A> {
Boolean equal<T  that>
}

Class Integer implements Comparable<Integer> {
private int value;
boolean equal (Integer that) {
return this.value == that.intValue();
}
}
```

2.1.4 Library Usage

The library user can use the provided ToUpload class in some application:

```
ToUpload files =  new ToUpload<Myfile> () ;
//generate some files- f1 and f2
files.add(f1);
files.add(f2);
files.uploadAll();
```

The reuse of the ToUpload container is quite intuitive. If the library use uses its own class for the type parameter Payload which does not implement getPath, then the compilation of this little code snippet will fail. The error message of the compiler will tell the user that his/her class does not implement the UploadAble interface. The getPath method is not provided by the user defined class.
Recursive type parameters can also be used by the library user, for example a list is supposed to be ordered is declared like this:

```
OrderedList<T extends Orderable<T>>
```

Even the library user can constrain himself in order to avoid bugs. In this case only subtypes of Orderable<T> can be used to create an OrderedList.This is ensured by the complier. An OrderedList can contain only elements which are subtypes of Orderable and a partial ordering exists on this collection. We use recursive bounds to avoid problems with covariance.

2.2. C++

2.2.1 Genericity internals

The type of translation of C++ templates and C++ concepts is called heterogeneous. For each instance of a generic class or function the compiler generated its own code. This type of translation can be seen as an advantage because the generated code is often faster compared to that of the homogeneous compilation method. When compiling heterogeneously the compiler can optimize the code much better especially for primitive types.

2.2.2 C++ Concepts

Current C++ templates system is fragile. It does not provide any type system for constrained type parameters. It can't check whether concrete types meet requirements when instantiated with type parameters. Library development and usage suffer because these contexts are not divides. Any time a user inserts a type which does not meet all the requirements as expected by the generic type, and then error messages combine the context of library development with that of usage. An upgrade of C++ template to C++ concepts is supposed to eliminate all these disadvantages. In this section a generic find function will be extended with C++ concepts. This example will give an introduction on how C++ concepts are used and what they offer.
C++ concepts will provide three different means to constrain generic types. Here is a short description of their purposes:

concept: This is an abstract interface-like collection of functions, operators and associated types. If a type is supposed to meet the requirements of a particular concept it must provide all the functions and operators ad specified in the concepts. An associated type for functions and operators within the concept in order to determine their types.

where: Where clauses constrain the type parameter in terms of a particular concepts. A concrete type must meet that concepts in order to be correct substitution for a type parameter. An experimental version of g++ which supports concepts uses requires clauses instead of where clauses.

concept_map: It specifies how a type meets the requirements of a concept. It maps the type into the domain of this particular concept. A concept_map can be template too.

2.2.3 Library development

Consider a fragile STL find function using templates.

```
template<typename InputIterator,typenameT>
InputIterator
find(InputIterator first,InputIterator last,const T& value) {
while(first<last && !(*first = =  value) )
++first ;
return first;
}
```

The user makes two calls to find with the appropriate types, according to the convention of the STL:

```
std : : vector<int> v;
find  (v.begin(), v.end(), 5); //okay
std : : List<int> l;
find(l.begin(), l.end(), 41); // error
```

Both vector and list iterators are InputIterators.The reason why finds fails to complie with list iterators is because they do not provide a < operator. The less-than-operator is part of the termination condition of the while loop within the implementation of this find algorithm. When compiling this code we get an error message saying that in the implementation of this find algorithm we don't have a less-than-operator.

In this section we will show how C++ concepts are used to create "better" generic functions. First we must create an InputIterator concept. The first and second arguments of find are supposed to meet the requirements of the InputIterator concept. When creating this concept, all functions and operators an InputIterator must provide are part of this concept, those are the increment (++) and the dereferencing (*) operator. The dereferencing operator returns an instance of a type which is determined by the iterator itself. When dereferencing an iterator (iterators are pointers) which points to list<int> it returns an instance of a different type than an iterator which points to an element of a different type. That's what the associated types are used for, in our case the associated type value_type is changed depending on the type of iterator. The associated type difference_type determines the distance between the begin and end of the iterator. This type is also iterator dependent. When iterating over an array of integers this distance is an int.This type varies as the iterator which is used. All we know so far is that the difference_type has to meet the requirement of a SignedIntegral type. This nested where clause states how this associated type difference_type has to behave. It must be an integer like type can be positive and negative. This is a complete InputIterator concept which states all requirements on types:

```
concept InputIterator<typename Iter> {
typename value_type;
typename difference_type;
where SignedIntegral<difference_type>;
Iter& operator++(Iter&);
Iter operator++(Iter&,int);
bool operator= =(Iter,Iter);
value_type operator*(Iter);
};
```

Futhermore, we need an EqualityComparable concept which ensures that types are comparable using equal. In terms C++ they must overload the = = operator. It also ensures that they have the same type.

This is our extended find version:

```
template<typename Iter , typename T>
where InputIterator<Iter>
&& EqualityComparable<InputIterator<Ite>: :
value_type,T>
Iter find (Iter first,Iter last,const T& value) {
while(first !=last && !(*first = = value) )
++first;
return first; }
```

The most interesting line from the point of C++ concepts is the where clause (if removes, the implementation is equivalent to STL's find, this is because of backward compatibility of templates and concepts).It says" The type of the type parameter Iter must meet the requirement of the InputIterator concept and the type of dereferencing operator InputIterator<Iter>: value_type must be equal-comparable with the type of the type parameter T".T is the type we are looking for.
In terms of C++ concepts we are not yet finished. We must establish a mapping between concrete types and the InputIterator concept. This is done because otherwise the compiler does not know which types meet this InputIterator concept.To establish a mapping between int* and the InputIterator concept we can declare this concept map:

```
concept_map InputIterator<int *> { . . . };
```

But this is not satisfactory because every pointer meets the requirements of an InputIterator concept. Therefore a concept_map can be template:

```
template<typename T>
concept_map InputIterator<T*>{
typedef T value_type;
typedef ptrdiff_t difference_type;
};
```

The body of a concept_map states how a type meets the requirements of a concept. In this case associated types value_type and difference_type are given concrete types which are determined by the type variable T.

2.2.4 Library usage

The library is now ready for usage, the library user uses the new find function:

```
int main() {
List<Person> persons;
persons.push_back(p1);          /*p1-p3 are instances of
Person*/
persons.push.back(p2);
persons.push_back(p3);
find(persons.begin(),persons.end(),person_looking_for);
}
```

The compiler ensures that the user uses only types which meet all the requirements of find.

3. CONCLUSION

This paper presents how generic programming is implemented in Java and C++.Java is a language where the current system for generic programming distinguishes between context of library development and the context

of library usage. The compiler diagnoses the error .In the correct context if some bug appeared somewhere in one of these contexts. Java is efficient and user friendly when developing and using generic libraries.C++ templates are fragile. An upgrade from templates to C++ concepts is supposed to eliminate these fragilities. Although this upgrade is not part of C++ standard yet, the expectations are quite promising.

REFERENCES

[1] A.Andrei. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison- Wesley Professional 2001
[2] R.Garcia, J.Jarvi, A.Lumsdaine, J.G.Siek and J.Willcock.A comparative study of language Support for generic programming
[3] B.Stroustrup.The C++ Programming Language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[4] Bjarne Stroustrup.Parameterized types for C++.Journal of Object-Oriented Programming, 1(5):5-16, 1989
[5] Joseph A.Bank, Barbara Liskov and Andrew C.Myers.Parameterized Types and Java. Technical report MIT LCS TM-553
[6] Luca Cardelli and Peter Wegner.On understanding types, data abstraction and polymorphism.
[7] Brian Cabana, Suad Alagic and Jeff Faulkner. Parametric polymorphism for Java: is there any Hope in sight? SIGPLAN Not.,39(12):22-31,2004.
[8] David vandevoorde and Nicoli M.Josuttis.C++ Templates: The Complete Guide.
[9] Sun microsystems inc.-a tutorial to generics. http://java.sun.com/docs/books/tutorial/generics/index.html.
[10] Concepts: Linguistic Support for Generic Programming in C++.Douglas Gregor, Jaakko J¨arvi, Jeremy Siek.
[11] A Comparative Study of Language Support for Generic Programming. Ronald Garcia Jaakko J¨arvi Andrew Lumsdaine, Jeremy Siek Jeremiah Willcock.
[12] M.Abadi and L.Cardelli. On subtyping and matching.

Shilpa Mathur has done Bachelors of Engineering in Information Technology from Rajasthan University, India in the year 2005.She has experience of working as a Lecturer in an Engineering College. Currently she is pursuing Masters of Technology in Software Engineering from Bhagwant University, India.