

# Comparison and Analysis to Extending a Existing Programming Language to Support Generics

Shu-Chun Weng  
Computer Science and  
Software Engineering Department  
National Taiwan University  
No.1, Sec. 4, Roosevelt Road,  
Taipei, Taiwan 106  
scw@csie.org

## ABSTRACT

The Java Programming Language and C#, Common Language Runtime made an extension to include generics or so called parametric polymorphism. These two languages are great examples for adding new feature to a already widely used language and many considerations make the task not so easy. There were a lot of proposals giving suggestions on it. We collect the representative ones and give comparison between them. We also analysis the final choose of Java deeply to see the advantages and shortcomings of it.

## 1. INTRODUCTION

Generics, parameterized type, or parametric polymorphism, allows developers of object-oriented programs to define generic classes or methods which can be used with similar but different interfaces respect to the instances to meet specific requirements. Generics has been used in C++, Eiffel, ADA and many other languages for a long time.

From the first publish of the Java Programming Language, JDK 1.0, to JDK 1.1 which introduced inner classes and reflections, through JDK 1.4, which includes several more language features, the generic mechanism was missing. C# Programming Language was introduced by Microsoft and is very similar to Java: compile to byte code instead of native binary, pure object-oriented and also lack of generics.

For a long time, Java and C# developers have to deal with the “Object” type (lowercase *object* in C#) which is a super type of any classes, and cast it to the already-known-correct subtype every time a generic collection or algorithm has been used. What’s worse, when one made a mistake, the cast may fail, which shall not happened when a generic collection implemented with parameterized type is used.

The generics extension to Java Programming Language (shorten

as “Java” below) has been added into JDK 1.5 [9]. Before then, many proposals and implementations have been proposed. The .NET Framework 2.0 with some new syntax to C# also supports generics based on the existing language syntax.

This paper will focus on different proposals and implementations as well as the final specifications of JDK 1.5 and C# 2.0. We shall examine those thought from the view of functionality, language tool chain, and developing.

The rest of this paper is organized as follows. We first briefly introduce several proposals and implementations in section 2. In section 3 we compare those ideas from different point of views. In section 4, we examine the Java 1.5 implementation more deeply. We point out the changes been made and the some suggestions which may make the implementation more smooth.

## 2. DIFFERENT PROPOSALS AND IMPLEMENTATIONS

By the time of '97 to '98, many ways of adding generics to Java have been proposed or implemented, including Pizza [13], Virtual Types [16], PolyJ [3], Agesen, Freund and Mitchell [1], GJ [5] and NextGen [7]. Later generics is added to Java and C#, and the details are presented in JSR-14 [4] and Kennedy and Syme [10]. In this section we shall briefly introduce proposals stated above with the *template* in C++ as compare. And for each approach, a small snip of sample code is given.

This is not a comprehensive list. For instance, MyT [2] and two approaches introduced in [15] are not covered since the target of such implementations are not the interesting part of this paper.

### 2.1 C++

The *template* keyword is added into C++ in order to support generics. It can be used to make generic version of classes, structures (the same as class), methods, or functions. The parameter of generic classes or functions can be compile time constants (including integers, pointers or references; language specification currently excludes floating point number here but there’s no technical difficulty on dropping such restriction) as well as types. No constrain can

be added to the parameter type and no constrain is needed since the type-relative checking is postponed to instantiate time when all type is determined.

```
template<typename T>
class SortedList {
public:
    void insert(const T& a) {
        T b;
        ...
        if (a < b) { ... }
    }

    template<typename Iter>
    void insertRange(Iter begin, Iter end);

    bool contains(const T& a);
};

SortedList<int> sorted_int;
sorted_int.insert(1);
```

## 2.2 Pizza

Pizza [13, 12] is a large project with a public available implementation. The target code is valid Java 1.4 code, that is, the new features are “translated”. It enhances Java with three major functionalities: generics, higher order function and algebraic types. However, the generics part becomes the base of the current officially chosen way and the other two are dropped. No keyword is introduced for genericity in Pizza, only angle brackets are used before *class* or return type of methods. The parameters should be types and can be bounded by *implements*. Since heterogeneous translation is used only for base types and homogeneous translation is used else where, the *implements* constrain is necessary for type-relative compile-time checking.

```
// This example is also valid AFM, GJ, Java 1.5
// and NextGen
interface Comparable<T> {
    boolean lessThan(T o);
}

class SortedList<T implements Comparable<T>> {
    public void insert(T a) {
        T b;
        ...
        if (a.lessThan(b)) { ... }
    }

    public void insertRange(Iterator<T> begin,
        Iterator<T> end);

    public bool contains(T a);
}

class OrdInt implements Comparable<OrdInt> {
    private int v;
    public OrdInt(int i) { v = i; }
    public boolean lessThan(OrdInt o) {
        return v < o.v;
    }
}
```

```
    }
}

SortedList<OrdInt> sorted_int;
sorted_int.insert(new OrdInt(1));
```

## 2.3 Virtual Types

This is a totally different approach from Pizza which has similar syntax and semantic to C++, but has been used in BETA [11] and ADA95 [14]. The type variable is defined as an inner type as *typedef* in C++, but the definition is virtual, that is, can be overridden by subclasses. However, in this approach, any instance of the class has to be an explicitly defined class instead of automatically generated by compiler when being used. And the parametric polymorphism can be applied only on classes but not methods.

```
interface Comparable {
    boolean lessThan(Object o);
}

class SortedList {
    typedef T as Comparable;

    public void insert(T a) {
        T b;
        ...
        if (a.lessThan(b)) { ... }
    }

    public void insertRange(java.util.Iterator begin,
        java.util.Iterator end);

    public bool contains(T a);
}

class OrdInt implements Comparable {
    private int v;
    public OrdInt(int i) { v = i; }
    public boolean lessThan(Object o) {
        if (o instanceof OrdInt)
            return v < ((OrdInt)o).v;
        else
            return false;
    }
}

class IntSortedList extends SortedList {
    typedef T as OrdInt;
}

IntSortedList sorted_int;
sorted_int.insert(new OrdInt(1));
```

## 2.4 PolyJ

Also known as *where clause*, this approach uses syntax similar to C++ with angle brackets replaced by square brackets and we can add some *where* clauses to bound the type parameters. The *where* clauses can contain *extends* or *implements* conditions and in-place-defined interface. Their implementation involves extending JVM, bytecode and verifier. Two opcodes (**invokewhere** and **invokestaticwhere**) are

added to call methods of where clause-bounded types. This implementation did not be chosen as official Java extension but the concept appears in the C# extension.

```
class SortedList[T] where T { boolean lessThan(T o); } {
    public void insert(T a) {
        T b;
        ...
        if (a.lessThan(b) { ... }
    }

    public insertRange(Iterator[T] begin,
        Iterator[T] end);

    bool contains(T a);
}

SortedList[Integer] sorted_int;
sorted_int.insert(new Integer(1));
```

## 2.5 Agesen, Freund and Mitchell

This approach has similar language construction as in Pizza but the implementation is different. They wrap the original .class file with a header describing the parameter count and constrains. Then the loader is changed to accept the header and when an instance of parameterized class is requested, the loader create the new class with the actual parameter types. Some bytecode are also changed to accept some “place holder” which will be replaced by actual type when loading. The “place holder” appeared in bytecode is used in C# extension but in that case, it is really used by the virtual machine instead of being replaced. Another notable point of this implementation is that it allows multiple constrains, e.g. an *implements* and an *extends*, on a single type variable which is a addition to Pizza and is accepted later. This approach will be referred as “AFM” afterward.

Sample code as in section 2.2

## 2.6 GJ

GJ is the winner of the proposals Java introduced in this section. The Java 1.5 altered it at some backward compactability issue. It adapts from the generics part of Pizza adding type inference rule, covariance and raw types. The “retrofitting” technology provides a simple way to translate legacy library which reduce an order of magnitude of work. On the implementing side, GJ compiles the extended Java code directly to Java bytecode so that the JVM don’t have to be changed. An attribute “Signature” is added so that the compiler can retrieve information without source file but it is totally ignored in run time.

```
// This example is also valid Java 1.5 and NextGen
interface Comparable<T> {
    boolean lessThan(T o);
}

class SortedList<T implements Comparable<T>> {
    public void insert(T a) {
        T b;
        ...
    }
}
```

```
        if (a.lessThan(b)) { ... }
    }

    public <Iter implements Iterator<T>>
    void insertRange(Iter begin, Iter end);

    public bool contains(T a);
}

class OrdInt implements Comparable<OrdInt> {
    private int v;
    public OrdInt(int i) { v = i; }
    public boolean lessThan(OrdInt o) { return v < o.v; }
}

SortedList<OrdInt> sorted_int;
sorted_int.insert(new OrdInt(1));
```

## 2.7 Java 1.5

As mentioned above, the generics part of the final specification of Java 1.5 is based on GJ. In fact, all four authors of [5] are listed in the JSR-14 document [4]. The additions are wildcard type variable [17], multi-bound and lower bound on type parameters. More details are stated in section 4.

Sample code as in section 2.6

## 2.8 NextGen

NextGen is another approach based on GJ. It’s a heterogeneous translating extension without changing virtual machine and focus on the weakness of GJ which cannot create an instance or an array of types given in parameters. Also, casting to such types and to an instance of parameterized type, which is not allowed in GJ or Java 1.5, are legal in NextGen. However, such features are chosen to be dropped from official specification currently because of the complexity and it’s hard to make it compactable with legacy code.

Sample code as in section 2.6

## 2.9 C# 2.0

The first version of C# and .NET framework was released in 2002 without generics support. However, in 2001, Kennedy and Syme introduced [10] the possibility to integrate generics into C# and .NET framework. In 2004, the formalization of it was published [18]. Then C# 2.0 was introduced in 2005 [8]. It is very likely improved from PolyJ. The parameterization can be applied on classes, methods, structures and delegations (anonymous methods). And the parameter type constrain has the form of *where clause* as in PolyJ but disallowing in-place-defined interface, adding “*class*” or “*struct*” and constructor constraint. The CLL, which is the bytecode of .NET Framework, has been extended to hold generics construction and the virtual machine is changed to handle it.

```
interface IComparable<T> {
    bool lessThan(T o);
}

class SortedList<T>
    where T : IComparable<T> {
```

```

public void insert(T a) {
    T b;
    ...
    if (a.lessThan(b)) { ... }
}

public void insertRange<Iter>(Iter begin, Iter end)
    where Iter : Iterator<T>;

public void bool contains(T a);
}

struct OrdInt : IComparable<OrdInt> {
    private int v;
    public OrdInt(int i) { v = i; }
    public bool lessThan(OrdInt o) { return v < o.v; }
}

SortedList<OrdInt> sorted_int;
sorted_int.insert(new OrdInt(1));

```

### 3. COMPARISONS OF PROPOSALS AND IMPLEMENTATIONS

Solorzano and Alagić [15] did some comparisons in their paper introducing a reflective solution. In this section we shall extend their work to compare more proposals with more view.

#### 3.1 According to the functionality

This criteria includes:

- Parametric polymorphism: instantiation parameters can be of any type and the checks based on parameter types will pass.
- Bounded parametric polymorphism: Each actual type variable has a non-generic upper bound.
- F-bounded parametric polymorphism: The type of an actual parameter has a bound that may be generically and recursively defined [6].

Obviously, only C++, which uses “text substitution”, satisfies parametric polymorphism. Virtual Types belongs to bounded parametric polymorphism where the bound is given but should be a static type. All others are F-bounded parametric polymorphism where the type parameters can be used as parameters of bounds.

It can be shown that meta-programming, which is a widely used programming technique in C++, cannot be used in generics supports only F-bounded parametric polymorphism. Java 1.5 specification clearly prohibits such feature.

#### 3.2 From the perspective of instantiation

This is distinguished by “the time different instances of a parameterized class are stored or considered different.” Under such definition, the categories includes:

- Compile, link time (heterogeneous): In the executable (or bytecode), multiple instances are stored.
- Load time (heterogeneous): Only single definition is stored but multiple instances are created at load time or by loader when needed.

- Never (homogeneous): All the instances are in fact the same after compilation, that is, the parameterization belongs to language level.

C++ is unquestionably instantiate at compile or link time depend on toolkits. Pizza handles base types separately and instantiates instances with base types as parameters at compile time. Virtual Types cannot choose approaches other than the first one because of the instances are always defined explicitly. NextGen is the only Java implementation with automatically generating parameterized classes which uses the first approach.

AFM instantiates classes by loader. All other Java extensions (including Java 1.5 and the usual part of Pizza) belongs to the third category. Implementation of C# 2.0 is not published but from the action we may assume it does the same as AFM, that is, loader create different classes but shares the bytecode.

#### 3.3 From the perspective of compiler and linker design

Unlike in [15], we doesn’t consider persistence store so that the categories remain are:

- Textual substitution: The source code of generic classes must be available. It is used as a kind of template or macro definition.
- Code substitution: The source code of generic classes is not necessarily available to the compiler.

C++ is the only one uses textual subsection and by now, no compiler implementation supports separate compilation which belongs to code substitution. All other approaches belongs to code substitution.

It’s not a coincident. The textual substitution is unsuitable for Java and C# since the source code of classes is not required to be available (especially commercial components) and none of the two provides “header” files.

#### 3.4 System support

Similar to the previous two sections, this is decided by “the time genericity is still known”:

- Static: Only the compiler is aware of parametric polymorphism (every compiler do).
- Semi-dynamic: The compiler and the loader are aware of parametric polymorphism.
- Dynamic: Not only the loader, but the run-time system is designed with parametric polymorphism in mind.

The categories are closely related to section 3.2. C++, Virtual Types and NextGen belongs to static support. However, NextGen supports covariant from GJ, such feature usually needs support by runtime, but fortunately the method invocation in Java bytecode includes more information than needed when not supporting covariant, the redundancy makes it possible to support it without changing the virtual machine.

Then, as in section 3.2, AFM belongs to semi-dynamic. But for C#, which also belongs to the load time heterogeneous in section 3.2, belongs to the dynamic category since the “place holders” are not replaced by actual types by loader but remains the same, then the virtual machine itself takes the responsibility for parametric polymorphism. Also, PolyJ, which uses homogeneous instantiation, belongs to dynamic because of the newly added bytecode as mentioned in section 2.4.

All other Java extensions, which belong to homogeneous in section 3.2, are statically supported. All the genericity are translated to unchanged bytecode set or unchanged Java source code and is hidden from runtime.

### 3.5 From the perspective of memory usage

Again, we use some changing points as standard in this section. To measure how memory efficient is an approach, we look at “the storage where duplication appears.” Note that if duplication appears in secondary storage (disks), so do in primary storage (memory), but not vice versa.

- Extensive secondary storage: Additional code is produced for every instantiation.
- Extensive primary storage: Complete instantiated classes are produced for every instances.
- Moderate primary storage: Inherited, alias, or instantiated class information objects are produced for each instance (the code is never duplicated).
- Space efficient: No in-memory object or files corresponding to instantiated classes are produced.

Again, these categories are close related to section 3.2.

C++ with no doubt goes to the first one. The C++ template is just a saver and more intelligent macro, code is repeated again and again. Both Virtual Types and NextGen should belongs to the first one as well, but they are not as storage consuming as C++ since no actual working code is saved. Only prototypes and some transferring code applying code reuse are stored and they are relatively small comparing to the real working code.

AFM belongs to the second one since the working code is copied and the type place holders are replaced. All other Java extensions are space efficient by that parameterization are erased by compiler.

C# goes to the third category since they support correct reflective information and such information have to be generated for every instances. There is an implementation called MyT [2] also belong to it.

### 3.6 Backward Compactability with Legacy Code and Library

The problem has two directions: legacy code uses modern library and modern code uses legacy library.

The biggest problem for legacy code to use modern library is on containers. For legacy source code or compiled legacy

binary, only Java 1.5 provided a complete solution so that everything goes well (when it doesn’t use reflection). With some care, Virtual Types can also have a compactable interface. Other approaches are theoretically able to do so with additional instance for backward compactability.

The problem for modern code to use legacy library also occurs on containers. A parameterized and instantiated container is not always suitable to be pass to legacy processing code. Again, Java 1.5 does the best since the introduction of “raw type” by GJ. But GJ itself without multiple bound on single type parameter makes the translation not always smooth. Virtual Types has all instances being subtypes of the original parameterized type, it can “trick” the legacy code as well. Solutions for other proposals are problematic.

To compare the compactability, the level is the work needed on converting legacy library code to modern one with caring backward compactability stated above:

- No change needed: This is just a dream.
- Slight changes or little works: Most of the original source of the library remains unchanged.
- A totally rewrite: Including processing and translating source tree.

C++ is this time not discussed. Pizza is not considered either since only one third of the changing is generics.

Virtual Types, PolyJ and AFM need a half automatic translation to replace the “Object” or similar interface name by type variables. Also, library of C#, .NET Framework, is an “*object*” everywhere collection and need a care translation.

GJ introduced “raw types” and suggested “retrofitting” which is also valid for Java 1.5. Only prototypes are rewritten by designer, the compiler can then add the signature information to the class file, even no recompiling is needed. They claimed that the retrofitting of the JDK 1.2 collection class can be done with no single exception. NextGen can use the same approach but since it dropped raw types and totally changed the base library, they suggest to put the generic version of library under “generic.\*” packages.

### 3.7 Performance

The performance affected by generics is dominated by the additional levels of lookup for methods compare to virtual invocation. For example, generic improved the performance in C++ since generic method call is a static address function call but a virtual call needs a function pointer and involves one level of indirectness.

Pizza, GJ, Java 1.5, NextGen, AFM and Virtual Types does not affect performance since method calls are by default virtual calls and for parameterized polymorphism the level of indirectness is still one. In C#, generics does not affect performance, either.

But performance drops in PolyJ since the introduction of bytecode `invokewhere` and `invokestaticwhere`, which force virtual machine to maintain a mapping from method name to address instead of a simple V-table.

### 3.8 Type Parameter Bound Representation

This is an extension on section 3.1 with more detail. But such freedom on writing constraints is not always necessary.

C++ cannot and do not have to add constraint. In fact, programmer can add as many constraints as one want – in user’s mind. The second boring case is Virtual Types. In Virtual Types, the “parameters” are written as inner type-define and can add some “base class” or “implements interface” constraints. But the classes and interfaces should be non-parameterized ones or fixed parameter ones.

Pizza, AFM and GJ has similar restrictions on constraints. All of them can added “extends” or “implements” to restrict the type variable being inherit from or implements specific classes or interfaces. AFM and GJ allows two restrictions appears together. NextGen reduces the syntax so that only “extends” is needed.

Constraints in Java 1.5 is quite flexible compare to the above ones. Multiple interface implementation requirement can be combined with base class constraint, and by reordering the constraints, represent type of the erasure can be changed. More over, a “lower bound” can be specific which is power full when combining with “wildcards” and output containers.

PolyJ is maybe the most flexible bound one can think. Any method you want the type variable has can be added to the in-place-defined interface. And any class providing such public interface can be used as parameter in spite of the interfaces it implements. This also reduces the number of interfaces needed and the problem on using binary-only components.

C# has similar constraint writing rule as in Java but there’s no lower bound nor wildcards. Instead, it adds the “*class*” and “*struct*” constraint which restrict a type variable to be a class or a structure. And there is a “constructor constraint” which make creating instance of parameter types possible.

### 3.9 Reflection

This section examines the correctness of reflection system and the usage of type variable under those implementations. C++ provides little reflective mechanism and everything is determined at compile time so that all it provides are correct. Similarly, the type variables are determined at compile time so that it can be used at everywhere a simple type can.

Virtual Types has all the instance being declared explicitly so that the reflection system works well and the type variables have maximum usage, too. In AFM, authors didn’t mention the reflection but since the loader can save the parameter information when instantiating instances, it could be correct. The another main target of AFM is that it is able to use type variables everywhere. C# has similar stand point to AFM and, as designer declared, has correct reflection system.

In Pizza, GJ, Java 1.5 and PolyJ, the type variables are “erased” and cannot be reconstruct at run time. The reflective information is incorrect, but, as Java specification mentioned, the reflection system returns the information of

the “raw type” which is compactable with legacy code. Also, the erasure makes type variables disappears at runtime and cannot be used to construct new object, create array, or doing type-checking and type-safe casting.

The main target of NextGen is to preserve the correctness of reflection system. It is sound and can then provide type-checking, type-safe casting, object creation, and array creation. But such job are performed zigzag.

### 3.10 Miscellaneous Points

There are still some interesting points in each proposals.

- Virtual Types has all the instance of a parameterized class inherit from the original generic class.
- NextGen uses a lot of dollar signs in the name of compiler generated instances and has some problem with inner classes although the inner class support is theoretically done.
- Author of AFM said that the AFM implementation consists of approximately 500 lines of Java code and a ten line change to the default loader in the Java Virtual Machine. This might be the easiest implementation.
- Java 1.5 does not allow parameters (including the type variable and parameterized types) are not allowed in *catch* clause but are allowed in C# since such feature cannot be done without changing the class file format and JVM.

## 4. NOTABLE CHANGES TO JAVA COMPILER, VM AND LIBRARY FOR GENERICITY

The Java compiler and library have been totally modified for Java 1.5. And there are some sneak points introduced in this section.

### 4.1 Grammar

For the newly added feature, angle brackets can appears in more places. Unlike C++ where a starting angle bracket will appears after the *template* keyword or a already known template type or function. The angle brackets will appears after the class name while the class is not known as parameterized type in advance. Parameterized methods make things more complicated – angle brackets can appear everywhere in the class scope.

However, all these can be solved by rewriting language context free grammar. And when rewriting the grammar, the “double less than” problem, which puzzled C++ programmers for a long time, has been solved. The problem arise when the parameter of a generic type is a generic type. The code may becomes

```
Vector<Pair<int, double>> a;
```

Then the double “less than” is tokenized as a right shift instead of two angle brackets and leads compile error. The correct version in C++ is

```
Vector<Pair<int, double> > a;
```

An extra space is necessary.

However, by rewriting the grammar, the first code snip is correct. The grammar given in [9] does not allow this but in [4], the part of grammar affecting this is:

```
ClassOrInterfaceType ::= Name
                        | Name < ReferenceTypeList1

ReferenceTypeList1 ::= ReferenceType1
                   | ReferenceTypeList , ReferenceType1

ReferenceType1 ::= ReferenceType >
               | Name < ReferenceTypeList2

ReferenceTypeList2 ::= ReferenceType2
                   | ReferenceTypeList , ReferenceType2

ReferenceType2 ::= ReferenceType >>
               | Name < ReferenceTypeList3

ReferenceTypeList3 ::= ReferenceType3
                   | ReferenceTypeList , ReferenceType3

ReferenceType3 ::= ReferenceType >>>
```

However, in the real implementation, this complicate grammar is not used. The core code managing such situation is `com.sun.tools.javac.parser.Parser.typeArguments()`. After parsing the parameter list, it is going to accept a “GT” token

```
switch (S.token()) {
case GTGTGTGT:
    S.token(GTGTGT);
    break;
case GTGTGT:
    S.token(GTGT);
    break;
case GTGT:
    S.token(GT);
    break;
default:
    accept(GT);
    break;
}
```

That is, it checks the next token and replaces it with the one with one fewer “greater than.”

## 4.2 Covariant and Function Overloading

Covariant means that the return type of a method can be narrowed when a subclass overrides it. Such overriding is allowed in Java 1.5 but not before 1.4. The implementation is translating straight forward [5, 4]:

```
class C { C dup(){ ... } }
class D extends C { D dup(){ ... } }
```

Above is valid Java 1.5 code as well as GJ code. Both translation of it becomes

```
class C { C dup(); }
class D {
    D dup/*1*/(){...}
    C dup/*2*/(){ return dup/*1*/(); }
}
```

The Java code is invalid but the bytecode equivalent is valid. The `C.dup()` is overridden by `D.dup/*2*/()`. The point is that in Java bytecode, the function prototype includes the return type. So that the two “dup” in D are not ambiguous.

## 4.3 Library Conversion

In section 2.6 and 3.6, we introduced retrofitting firstly raised by [5]. Retrofitting is to write parameterized declaration for classes and methods separate from the source. Then, by providing the GJ compiler the original class file and the declaration with some special flags, the compiler will store the information in a attribute called “Signature.” The old code did not even need to be recompiled.

But, however, in JDK 1.5, especially under package `java.util`, the classes and interfaces has been totally rewritten to follow the generics programming convention. From another point view, this might mean that Sun Microsystems does not satisfy stopping here. Although Java 1.5 introduced generics in the the Java Programming Language successfully with a great backward compactability, the broken reflection system and type erasure are the pay off. A system solves such problems could not use retrofitting so that they really rewrite the whole library to prepare for the next step.

## 4.4 Raw Type

Raw type is a brand new idea appears in GJ. The great backward compactability is totally based on it. But the existence of raw types breaks the reflection system, which is a very important part in modern programming languages. After applying a series of type erasure, the raw type forms, and after the compiler translated the Java source code, all the instances of parameterized classes are replaced by the raw type. Which means that raw type is instead the base of generics in Java 1.5.

But, as pointed out by [15], *generic classes are not classes, but rather class functions*. The existence of raw type make the typing system collapse. They must have a separate representation, and be viewed just as a shared method repositories.

## 5. CONCLUSIONS

This paper firstly introduces nine different approaches on extending an existing language to support parametric polymorphism. C++ is included because of its well known implementation, advantages, and shortcomings. The second part is based on [15] with more point of view and more complete analysis on all kinds of implementation. Then sneak points of Java 1.5 implementation is discussed.

As an obvious result of the third section, Java 1.5 has a good rating in these proposals. The main and gravest weakness of it is the problem on reflection system and the disappearance of type variable which stopped many runtime checks.

A problem discovered by Bracha [5] is that “security channel does not secure”:

```
class SecureChannel extends Channel {
    public String read();
}
class C {
    public LinkedList<SecureChannel> cs;
}
```

Then C.cs got erased to have type LinkedList and no more checks can be performed at runtime. If an attacker write in Java bytecode adding a non-secure channel into it, the virtual machine could not detect it and the security crashed.

There are at least two approaches to fix the problem on reflection in Java 1.5 in [15], we hope that those ideas can be considered in the future version.

## 6. REFERENCES

- [1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the java language. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 49–65, New York, NY, USA, 1997. ACM Press.
- [2] S. Alagic, J. Solorzano, and D. Gitchell. Orthogonal to the java imperative. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 212–233, London, UK, 1998. Springer-Verlag.
- [3] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized types for java. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 132–145, New York, NY, USA, 1997. ACM Press.
- [4] G. Bracha, N. Cohen, C. Kemper, M. Odersky, D. Stoutamire, B. Thorup, and P. Wadler. Adding generics to the java programming language: Public draft specification, version 2.0. Technical report, Sun Microsystems, 2003.
- [5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.
- [6] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM Press.
- [7] R. Cartwright and J. Guy L. Steele. Compatible genericity with run-time types for the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 201–215, New York, NY, USA, 1998. ACM Press.
- [8] M. Co. C# language specification. <http://www.ecma-international.org/publications/standards/Ecma-334.htm>, June 2005.
- [9] G. S. James Gosling, Bill Joy and G. Bracha. *The Java Language Specification, Third Edition*. Java Series, Sun Microsystems, 2005.
- [10] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2001. ACM Press.
- [11] O. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
- [12] M. Odersky, E. Runne, and P. Wadler. Two ways to bake your pizza - translating parameterised types into java. In *Selected Papers from the International Seminar on Generic Programming*, pages 114–132, London, UK, 2000. Springer-Verlag.
- [13] M. Odersky and P. Wadler. Pizza into java: translating theory into practice. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159, New York, NY, USA, 1997. ACM Press.
- [14] E. Seidewitz. Genericity versus inheritance reconsidered: self-reference using generics. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 153–163, New York, NY, USA, 1994. ACM Press.
- [15] J. H. Solorzano and S. Alagic. Parametric polymorphism for java: a reflective solution. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 216–225, New York, NY, USA, 1998. ACM Press.

- [16] K. K. Thorup. Genericity in Java with Virtual Types. In *ecoop*, 1997.
- [17] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.
- [18] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .net common language runtime. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 39–51, New York, NY, USA, 2004. ACM Press.