Java ParTy: Java with Parameterized Types

Abstract

In Java, a programmer who wants to define a group of related classes that have similar behavior but manipulate different types cannot easily do so. In this paper, we propose the addition of type parameters to facilitate the construction of generic classes and interfaces. Our solution allows generic code to impose type constraints on the actual type used, so that certain guarantees about the actual type can be preserved. We then show how a working implementation of our design, Java ParTy, allows safe, generic Java code without changing the JVM bytecode or incurring any additional run-time type casting overhead.

1 Description of the Problem

In Java, a programmer who wants to define a group of related classes that have similar behavior but manipulate different types cannot easily do so. An ideal solution would allow the programmer to define a generic abstraction of the class, and then allow the class to adapt according to the type with which it is instantiated. This solution would require only one generic class to be written for a potentially large number of different instantiations, greatly reducing the amount of code that needs to be written. In addition, the ideal solution would still be able to guarantee type safety before run time, so it would not introduce any new type errors.

Unfortunately, Java does not provide any simple and efficient method to achieve this kind of genericity. Currently, a programmer that wishes to use generic code must design classes that use parameters of type Object. Through type casting, a particular parameter can be cast into Object, and manipulated by the class. This solution provides no mechanism for restricting the possible types used in the instantiated class, risks type equivalence problems, and incurs the heavy run-time overhead of type casts.

However, Java was intended to be extendable as the need for new programming power arises [BLM97]. We believe that Java users would benefit from the additional power of generic abstractions in code, and in our project, we have designed and implemented an extension to Java that adds support for parametric polymorphism via parameterized types.

There have already been several attempts at offering support for parameterized types in Java.
Each of these previous attempts is cleverly designed to offer complete support for all flavors of parametric polymorphism. Accommodating all possible scenarios of generic abstractions is a difficult programming task, and the current extensions handle these problems quite well.

Instead of trying to provide a better version of the same complete extension, we take a different approach. Firstly, we have examined what properties of parameterized types are most valuable to Java users. We wish to answer the question: What makes parameterized types valuable to programmers? We believe that determining the usefulness of the extension is of primary importance, and we will then use this knowledge to guide our design, rather than attempting to provide full support first, and then reflect on its usefulness after the implementation is complete.

After choosing what makes parameterized types valuable to programmers, we will finally design and implement a limited version of these features. We aim to add only the most useful extensions to Java, and will not add complexity to the language unless its usefulness outweighs its complexity.

2 Related Work

In our consideration of related work, we examined both the existing support for type parameterization in other languages, and the current work done on Java to support parameterized type extensions. We chose to examine the existing support in languages other than Java in order to deduce exactly what features of type parameterization were desirable ones. The languages that already support generic types generally have a large amount of code written that takes advantage of it. From examining this code, we were better able to decide which supported features could be considered useful, and which supported features were not. The languages under consideration included C++, Ada, Sather, Eiffel, Haskell and ML.

We chose to examine the currently proposed extensions so that we could better understand the unique challenges in adding type parameterization to the Java programming language. Although we reviewed a large amount of work in this area, we concentrated our efforts on examining PolyJ and GJ, two separately proposed extensions that are among the top proposals currently being considered by Sun Microsystems [Bra99].

2.1 Parameterized Types in Other Programming Languages

2.1.1 C++

C++ supports type parameterization through the use of templates. Templates were added to C++ in 1989 with the release of C++ 2.0, and were not a part of the original language design [Str93]. The parameter list of a template definition can include both type and value information. The type information allows a particular type to be passed into the corresponding function or class so that it can be properly instantiated for that type. The value information is the same as a function value parameter, and contains a value of a known type. Although value template parameters are supported by C++ templates, they are rarely used [CD99].

Templated classes in C++ are instantiated heterogeneously, so that a copy of each class exists for
each appropriate type that could be passed as a parameter to the template. This allows C++ to appear to have dynamic type binding when it actually provides static binding, and allows C++ to provide type parameterization without sacrificing code execution speed. However, it causes an increase in code size.

Templates do not impose any restrictions on what a type must provide in order for it to be an appropriate candidate as a type parameter. Thus, a templated class that attempts to use a method of a type for which the method does not exist can cause a run-time error. Other languages avoid this problem by including type constraints in the declaration of the templated class that specify necessary features of the candidate type.

Templates support primitive types as well as user-defined types as type parameters. In fact, a type parameter can be another templated class itself. A class nested inside another class can share the same templated type information. Also, declarations of a templated class can implicitly include type information, based on other parameters, and can avoid requiring the programmer to explicitly declare the parameterized type.

In general, C++ templates were added to provide a maximum amount of flexibility on top of an already existing language. Although flexible, some of the features of templates were excluded from our design decisions because we did not consider them necessary to genericity.

### 2.1.2 Haskell and ML

Haskell and ML both support the Hindley-Milner type system [Mil78], which naturally lends itself to polymorphic types and generic code. In such functional languages, we find that some polymorphic types are, in a sense, strictly more general than others. For example, the type \([a]\) is more general than \([\text{Char}]\). In other words, the latter type can be derived from the former by a suitable substitution for \(a\). With regard to this generalization ordering, Haskell's type system possesses two important properties: First, every well-typed expression is guaranteed to have a unique principal type, and second, the principal type can be inferred automatically.

An expression's or function's principal type is the least general type that, intuitively, "contains all instances of the expression." For example, the principal type of \(\text{head}\) is \([a]\rightarrow a\); the types \([b]\rightarrow a\), \(a\rightarrow a\), or even \(a\) are too general, whereas something like \([\text{Int}]\rightarrow \text{Int}\) is too specific. The existence of unique principal types is the hallmark feature of the Hindley-Milner type system, which forms the basis of the type systems of Haskell, ML, Miranda, and several other functional languages [Kre98].

The Hindley-Milner type system requires that every expression be properly typed, but the languages themselves do not provide methods for providing additional type constraints. Without extra type constraints, another user of a polymorphic function may be able to use an unsuitable type with the function, causing unexpected behavior from that function.

### 2.2 Proposals for Parameterized Types in Java

#### 2.2.1 GJ and Pizza
GJ, or Generic Java, was designed in 1998 by Gilad Bracha of JavaSoft, Martin Odersky of the University of South Australia, David Stoutamire of JavaSoft, and Philip Wadler of Bell Labs, Lucent Technologies. It was originally based on the handling of parametric types in Pizza [OW97], but uses a simpler type system, and provides greater support for backwards compatibility. All GJ code translates into normal Java virtual machine (JVM) code. The designers chose this method over modifying Java bytecodes because it preserves the safety and security of the Java platform, and would allow for forwards and backwards compatibility with previously existing Java code.

GJ programs look much like the equivalent Java programs, except they have more type information and fewer casts. The GJ compiler erases type parameters, replaces the type variables by their bounding type (typically Object), and adds the appropriate type casting. Thus, generic code written in GJ is translated into code for type Object, and casts are used to appropriately modify the code for each desired type. This method allows for homogeneous genericity, and avoids the code size "blowup" that occurs in heterogeneous implementations like the one found in C++ [BOSW98a].

The method used by GJ to support parameterized types could just as easily be done by programmers themselves. By allowing the GJ compiler to handle type casting instead of requiring the programmer to do it, GJ pushes the possibility of run-time type errors into compile time, adding to type safety. However, GJ is really a hack that does not change the underlying structure of Java to accommodate for parameterized types in any way. Because of this, GJ code is filled with run-time type casting, usually between the desired type and type Object. This causes a significant decrease in code execution speed.

2.2.2 PolyJ

PolyJ was designed in 1997 by Andrew C. Myers, Joseph A. Bank and Barbara Liskov of Massachusetts Institute of Technology. PolyJ is an extension to Java that supports constrained parametric polymorphism. It allows abstractions to be parameterized so a single piece of code can implement many related abstractions.

Based on previous work done in CLU [LSAS77], PolyJ provides the where clause as the mechanism for constraining parameterized types. The where clause allows the programmer to state that a class conforms to a certain constraint without explicitly declaring the relationship when the class is defined. For example, consider a hashtable class in which it is necessary to obtain a hashcode for each object inserted. A where clause allows the programmer to restrict the set of possible types to those that provide a hashcode method. This differs from other project such as Pizza and the following Stanford project, which rely on the programmer to use only those types that are appropriate for the generic class. Because the power and safety of the where clause has been proven to be effective in PolyJ as well as in CLU, we have chosen to implement this feature in our language extension.

In order to provide good performance for parameterized code in both execution speed and code size, PolyJ makes significant changes to the bytecodes and to the JVM. This was found to be necessary to allow shared code while keeping constant pool information separate among instantiations of a parameterized class. This makes the implementation more costly and the
virtual machine more complicated. Any change to the bytecodes and bytecode verifier will require all safety and security aspects of the system to be reevaluated. Furthermore, PolyJ code will not be backwards compatible with any normal JVM which may discourage the current Java programming population to use PolyJ.

2.2.3 Agesen, Freund, and Mitchell

This project [AFM97] was headed by Ole Agesen of Sun Microsystems Lab, and Stephen N. Freund and John C. Mitchell of Stanford University. It supports only the type relations implements and extends, which are used to extend a parameterized type for the Java programming language. This mechanism differs from the where clause constraint mechanism in that it cannot express constraints that involve the type parameter itself [BLM97].

Unlike other projects that change Java bytecodes, Agesen et al. inserts a preprocess step into the load phase of the JVM. By delaying instantiation until load time, they are able to achieve an appropriate balance between language expressiveness, run-time efficiency and compiled code size, without changing the language definition or making major modifications to the JVM. We anticipate that this method will be easiest for Java programmers to adopt and use effectively, and interfere minimally with any current or future developments in Java implementations. Because of these benefits, we have chosen to take the same approach in the ParTy implementation. Their implementation, however, lacks a type constraint mechanism, which means that users may not write safe, generic code.

3 Solution: Java ParTy

3.1 Language Design Criteria

ParTy aims to add only the most useful extensions to Java, and does not add complexity to the language unless its usefulness outweighs its complexity. We chose to model our design by combining the most desirable features of the Agesen et. al. load-time extensions with the power and safety of the PolyJ language design. Through this combination, we were able to provide a powerful construct for designing generic classes and interfaces without making any major changes to Java bytecodes or the JVM. ParTy intends to maximize programming potential without modifying the original Java specification in any destructive way.

3.2 Language Definition: Syntax Specification

The following subsections cover all ParTy additions to Java syntax, as specified in the Java Language Specification [GLS96].

3.2.1 Keywords

ParTy adds one new keyword to Java: the clause.

Keyword: one of
abstract default if private throw
3.2.2 Type Syntax

A parameterized type consists of a class or interface type and a parameter section. The name must be the name of a parameterized class or interface, and the types in the parameter list must match the number of declared parameters of , and each actual parameter must be a subtype of the formal parameter's bound type [GJ specification].

ParTy does not support parameterized methods outside a parameterized class, so a type variable cannot be introduced by a method alone. Therefore, we append the following to the type syntax of the Java Language Specification (JLS, section 4):

Type:
- PrimitiveType
- ReferenceType

(see JLS Section 4.1)

ReferenceType:
- ClassOrInterfaceType
- ArrayType
- TypeVariable

TypeVariable:
- Identifier

ClassOrInterfaceType:
- ClassType TypeArgumentsopt
- InterfaceType TypeArgumentsopt

ClassType:
- Identifier
- ClassType . Identifier

InterfaceType:
- Identifier
- InterfaceType . Identifier

TypeArguments:
- < ReferenceTypeList >
ReferenceTypeList:
  PrimitiveType
  ClassorInterfaceType
  ReferenceTypeList , ClassorInterfaceType

3.2.2.1 Examples

LinkedList<int> A; // primitive types are permitted
Vector<String> B;
Collection<Collection<Integer>> C; // A Collection of Integer-typed Collections

3.2.3 Class and Interface Declaration Syntax

A parameterized class declaration defines a set of types with constraints. One of these types is specified for each type parameter in the declaration when the class is instantiated.

ClassDeclaration:
  ClassModifiersopt class Identifier TypeParametersopt Superopt Interfacesopt
  ClassBody

InterfaceDeclaration:
  InterfaceModifiersopt interface Identifier
  TypeParametersopt ExtendsInterfacesopt InterfaceBody

TypeParameters:
  < TypeParameterList > Constraintsopt

TypeParameterList:
  TypeVariable
  TypeParameterList , TypeVariable

Constraints:
  where ConstraintList

ConstraintList:
  ConstraintArguments
  ConstraintList , ConstraintArguments

ConstraintArguments:
  TypeVariable Methodsopt TypeBoundopt

Methods:
  {MethodHeaders}

MethodHeaders:
  MethodHeader
  MethodHeaders , MethodHeader

(see JLS Sect. 8.4 for MethodHeader)

TypeBound:
  extends ClassType
  implements InterfaceType
  extends ClassType implements InterfaceType
### 3.2.3.1 Examples

```java
class simpleClass<T> {
    // class body goes here
}
```

```java
public class myConstrainedClass<Type1, Type2>
    where { boolean lessThan(Type1 t) } extends myBaseClass implements
myInterface {
    // class body goes here
}
```

### 3.2.4 Discussion

In general, ParTy adds support for type parameters within class or interface definitions, including the use of the where clause to impose additional constraints. A detailed discussion of design decisions influencing ParTy syntax follows.

We chose not to allow value information to be coupled with type information in the parameters, opposing C++-style parameterization. In our syntax, a reference type list can produce a type variable or another type parameter list followed by a type variable. Thus, it cannot produce any type identifier other than a type variable identifier. Encapsulating value information with type information adds no new power to the generic code; any value information can just as easily be passed as a functional parameter through a constructor. Rather, this encapsulation is harmful because it blurs the distinction between type and value parameters, and we prefer this distinction to be clear.

For example, this class declaration and constructor:

```java
class notPermitted<type, value> {
    type t;
    int v;
    public notPermitted<type, value>() {
        t = type;
        v = value;
    }
}
```

could have just as easily been written in ParTy as:

```java
class permitted<type> {
    type t;
    int v;
    public permitted<type>(int value) {
        t = type;
        v = value;
    }
}
```

and the latter version maintains a healthy separation of type and value parameterization.

We chose to allow multiple type parameters to be passed in a single instantiation of a
parameterized class. In addition, we chose to allow type parameters to themselves be of a parameterized type. Syntactically, a reference type list can itself contain another reference type list. This allows for F-bounded polymorphism [CCH+ 89], where a type argument can itself be a parameterized type. This is necessary for any non-trivial application of generic classes, and is why we included this feature. One could easily imagine a large-scale application that takes advantage of passing parameterized classes as type parameters.

This recursive definition of type parameters can cause a small lexical problem with the angle brackets used to specify multiply nested type parameters. For instance, the double angle brackets in Vector<Seq<String>> may be incorrectly identified as the right-shift symbol >>. We have omitted the extra syntactic additions required to accommodate this problem for brevity. For a study of one solution to this problem, see Bracha et al. [BOSW98b].

Parameterized declarations can declare subtype relationships. E.g., SpecialStack can be declared to be a subtype of Stack:

class SpecialStack[T] extends Stack[T] { … }

Such a subtype declaration is legal if the new subtype satisfy the constraints of the supertype. It should be noted that any instantiation of T should appear the same in SpecialStack[T] and Stack[T]. For example, SpecialStack[String] can be stated to be subtype of Stack[String], but it is illegal to state that SpecialStack[String] is subtype of Stack[Object] even though Sting is a subtype of Object.

Formally, an instantiation is never a subtype of another instantiation that differs from it in one of the actual parameters. The reason for this restriction is easy to understand. Consider the following code:

```
Stack<Object> b = new Stack<String> ();
b.push(new Object());
Stack<String> c = (Stack<String>) b;
String x = c.pop();
```

If the subtype relationship were allowed, the initial assignment to b would be legal. The push(Object ob) method call would also be legal. Furthermore, the cast would succeed since b is a Stack[String] object, and therefore the assignment to x would be executed. But this assignment would cause x to refer to Object element, which is not type correct.

The type arguments can optionally be followed by any number of constraint arguments. These constraints require a type variable to have an associated method that satisfies the corresponding method header defined for it. The constraint is implemented with the where keyword, which has been added to the set of Java keywords.

Basically, the information in the where clause serves to isolate the implementation of a parameterized definition from its uses (i.e., instantiations). Thus, the where clauses permit separate compilation of parameterized implementations and their uses, while still enforcing type constraints, which is impossible in C++ and Modula3. We have chosen to support the where clause in our implementation for such reasons.

In implementing where clauses, the compiler not only checks whether a type t has particular
method \( m \) declared in the where clause, but also checks the type of the arguments of \( m \). It verifies whether a hypothetical call to \( m \) will succeed.

Though CLU provided the ability of applying additional constraints to individual methods, the ability also adds substantial complexity to the design and implementation. We omit it in our implementation. The following example shows additional constraints that we will not support:

```
interface SortedList<T>
    where T { boolean lt (T t); } // ALLOWED: constraint is applied to
interface
    {
        ...
        void output(OutputStream s)
            where T { void output (OutputStream s); } // NOT ALLOWED:
additional method constraint
        (...)
    }
```

### 3.3 Language Definition: Semantics Specification

(1) \( PC[n] = ClassName < Type_1, Type_2, \ldots Type_n > Id; \)
& Check ( ClassName < Type_1, Type_2, \ldots Type_n > ) [parameterized-class-instantiation]

\[
\text{ClassName}_{\_Type_1}\_Type_2\ldots\_Type_n \text{ Id;}
\]

(2) Check ( ClassName < Type_1, Type_2, \ldots Type_n > )
& ClassModifiers opt class ClassName <Type_Var_1,Type_Var_2,\ldots,Type_Var_n >
    where Type_Var_1 Methods opt TypeBound opt,
    Type_Var_2 Methods opt TypeBound opt, ...
    Type_Var_n Methods opt TypeBound opt
    (ClassBody)
    & Check(Type_1 Methods opt TypeBound opt)
    & Check(Type_2 Methods opt TypeBound opt, ...)
    & Check(Type_n Methods opt TypeBound opt)
    [parameterized-class-check]

\[
\text{ClassModifiers opt class ClassName}_{\_Type_1}_{\_Type_2\ldots\_Type_n}
\{\text{ClassBody}( \text{Type}_1/\text{Type}_{\_Var_1}, \text{Type}_2/\text{Type}_{\_Var_2}, \text{Type}_n/\text{Type}_{\_Var_n}) \}^1
\text{return True}
\]
(3) Check( Type { Method_1, Method_2, ... Method_k }^2 extends A implements B )
& Type has Method_1
& Type has Method_2, ...
& Type has Method_k
& Type extends A implements B
[extends-implements-check]

return true

(4) Check( Type { Method_1, Method_2, ... Method_k }^2 extends A)
& Type has Method_1
& Type has Method_2, ...
& Type has Method_k
& Type extends A
[extends-check]

return true

(5) Check( Type { Method_1, Method_2, ... Method_k }^2 implements B )
& Type has Method_1
& Type has Method_2, ...
& Type has Method_k
& Type implements B
[implements-check]

return true

Footnote:
1 Create new class, where ClassBody( Type /Type_Var_1 , Type_/Type_Var_2 , ... Type_/Type_Var_n ) is produced by replacing every occurrence of Type_Var_1
Type_Var_2 ... Type_Var_n in ClassBody with Type_1, Type_2 ... Type_n respectively.

2 k >= 0

3.4 Implementation

Figure 1 shows a schematic diagram of the Java Virtual Machine. When a running program
refers to a class that has not been loaded, compiled bytecode passes through three largely
independent components of the virtual machine:

(1) The \textit{Loader} obtains the bytecode for the class from either a local file or a remote site;

(2) The \textit{Verifier} validates the bytecode by checking that operation codes are valid, branches are
to legitimate locations, methods have structurally correct signatures, and that every instruction
obeys the Java type discipline;

(3) The \textit{Linker} initializes static fields of the new class and may load related classes.

![Diagram of Java Virtual Machine](image)

Figure 1: Java Virtual Machine

It is possible for Java programs to modify the behavior of the loader on certain classes by
extending the Java ClassLoader class. For example, we may change the loader to preprocess the
class before allowing the default loading mechanism to deal with the class. This is the basic idea
behind our ParTy language extension.

Our general strategy was to compile a parameterized class into an extended form of the class file.
In addition to all the information usually found in a class file, the extended file includes
information about type parameters and constraints. When the virtual machine attempts to load an
instantiation of a parameterized class or interface, a loader preprocess phase transforms the
parameterized class file into the particular desired instantiation and then declares it as if it were a
normal class. For example, the contents of the class \texttt{stack<T>} may be used by the loader to
heterogeneously generate both \texttt{stack<Integer>} and \texttt{stack<String>} classes if needed.

The following subsections describe our ParTy program model and corresponding ParTy class file
format that extends from general Java Class File format.
3.4.1 ParTy Program Model

Figure 2 illustrates the basic mode of ParTy program translation into JVM-legal class files:

(1) ParTy programs are compiled by a ParTy compiler:
   - For a Java source code of parameterized class or interface definition, the compiler takes the parameterized code, and compiles it into a ParTy class file, i.e., a parameterized class.
   - For programs which create instances of a parameterized class, the compiler compiles them into ParTy class files that support parameterized class information in the class file format.

(2) Before the JVM gets ready to run a ParTy program, our load-time preprocessor creates instantiations of parameterized classes. The steps to load an instantiation are:
   - Convert the name of the class into a parameterized class name and argument names. For example, an instance of class Stack<String> will have name Stack_String.
   - Translate the argument names to class and interface types. This may involve loading other class files. For example, class Stack<specialString> where specialString is a user-defined class will invoke loading of class specialString.
   - Find the file containing the binary representation of the parameterized class and read it into memory.
   - If the instantiation is valid, generate the binary representation of the instantiation from the parameterized class file and arguments; proceed in the usual manner to create a Class object for it.
3.4.2 ParTy Class File Format

A typical Java class file is a binary representation for a compiled class that can be loaded by any Java Virtual Machine. The constant pool contains strings representing all names mentioned in the class file, including class names, field and method names, type names, and so on. One convenient aspect of the class file format is that most type information generated when a class is compiled is stored in its constant pool as strings. The only exception is type information about primitive types [AFM97], which is embedded directly into the bytecode instructions.

Therefore, by extending the special strings format in the constant pool, we have created the ParTy class file format, containing parameterized types and their constraint information (see Fig. 3).

![Figure 3: ParTy Class File Format extensions for parameterized classes.](image)

Specifically:

- When a parameterized class is defined, the Utf8 string in the constant pool, i.e., `<classname>` will be changed to `<classname>_ParTy_<type>(<_type>)*`, where `classname` is the class name. String "ParTy_1_<Constraints>", "ParTy_2_<Constraints>", and so on, will be used for each type parameter and will appear wherever it is supposed to be if it is the actual class. For example, the parameterized class `specialStack<T>` where `T` has method1, will have the corresponding `<classname>` in the constant pool `stack_ParTy_ParTy_1_has_method1`;

- When a parameterized class is used, i.e., when a program (another class) creates a parameterized class, the Utf8 string in the constant pool, i.e., `<classname>` will be changed to `<classname>_ParTy_<type>(<_type>)*`, where `classname` is the class name and each type is an actual type parameter appearing in the definition of the ParTy source code. For example, for the following pjava code:
stack<String> st = new stack<String>();

the parameterized class file for the Utf8 string "stack" will be changed to: "stack_ParTy_String ".

In the load-time preprocessing phase, a parameterized class is instantiated into different instantiations. The algorithm for transforming a parameterized class file into binary representations is described in Section 3.4.1.

### 3.4.3 Discussion

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Language Extension</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>No constraints</td>
<td>Heterogeneous; instantiated at compile time</td>
</tr>
<tr>
<td>PolyJ</td>
<td>Where clauses</td>
<td>Homogeneous; separate type info.</td>
</tr>
<tr>
<td>GJ/Pizza</td>
<td>Implements/extends</td>
<td>Homogeneous; shared type info.</td>
</tr>
<tr>
<td>ParTy</td>
<td>Where clauses</td>
<td>Heterogeneous; instantiated at load time</td>
</tr>
</tbody>
</table>

Figure 4. Summary of the language extensions and implementation techniques

Several advantages can be achieved by delaying instantiation until load time. Firstly, the bytecode verifier and interpreter remain unaffected, and thus the type correctness and other properties of Java program execution are not compromised. Secondly, the ParTy class file format allows for separate compilation of a parameterized class and programs which create instances of it. Once the parameterized class file has been generated, a check against the constraints stored in that file guarantees type correctness of an instantiation. Thirdly, load-time instantiation provides size and speed tradeoffs. The existence of only a single compiled class file for all instantiations of a parameterized class makes binary representations compact on disk and allow efficient distribution of Java classes over a network.

The heterogeneous approach to generating typed classes used in our load-time preprocessor has some disadvantages also. Compared to a homogeneous implementation, memory demand on the system increases because a new class is generated for each instantiation. The increase is proportional to the number of differently-typed instantiations of the parameterized class. We may expect that the memory increase will be small relative to the memory requirements for the system as a whole. ParTy load-time instantiation of parameterized classes also increases the time
spent on loading classes during program execution. The heterogeneous class production approach enables faster execution because the resulting classes often contain fewer run-time type checks, and we chose it for this reason.

Because the code blowup caused by the heterogeneous class file production does not occur until load time, generic ParTy code does not require extra space when compiled. This means that compiled ParTy code is equivalent in size to the corresponding Java code. In fact, it may be smaller, as the ParTy code does not need to contain any extra type casting. This property makes ParTy code ideal for transmission over a network such as the Internet, where the code size is still a major concern. The delay in code blowup allows small, efficient, generic code to be sent quickly without penalty.

### 3.4.4 Experimental Implementation

We divide the implementation work into two parts: (1) a ParTy compiler, and (2) the load-time preprocessor. So far, a prototype of the JVM load-time preprocessor has been implemented and tested using Sun’s JDK 1.1.5 source release. While a full compiler for parameterized classes has not yet been implemented, this prototype is sufficient to test the effectiveness of load-time instantiation. Figure 5 presents a detailed illustration of the ParTy class translation in our experimental implementation.
We developed a special prototype compiler to compile our ParTy java source code to get ParTy class file. This special compiler will analyze ParTy source code, rewrite its parameterized class names into a valid syntactic form allowed by the normal Java Language Specification, create the necessary dummy classes for type variables, and then pass them to a normal JDK compiler. In this way, a ParTy class file is produced.

When instantiating a ParTy class at preload time, load-time preprocessor will:

- Parse ParTy class file’s constant pool;
- Find out special strings that is an instance of a parameterized class;
- Produce corresponding binary representation for this specific instantiation;
- Perform special string replacement to get a normal JDK class binary representation.

A subclass of the ClassLoader should be built in order to integrate the ParTy load-time
preprocessor into JVM and use it automatically. However, it is restrictive for general use since we could not install that `ClassLoader` as the default for the JVM. In our current implementation we left the `ClassLoader` unchanged, which does not change our evaluation of the load-time instantiation performance.

Instantiation with primitive types is another implementation issue we did not implement in current version. There two ways to implement it: Wrapper class or Modify bytecode directly. Future work can add primitive type arguments into it.

### 3.4.5 Some detail of Load-time preprocessor implementation

The following is how our experimental load-time preprocessor goes:

- Extract the class name and corresponding parameters from the bytecode (class file) of the program that creates instances of a parameterized class;
- Find the pclass, instantiate it using the parameters obtained in 1);
- The program (name it as B) that uses parameterized classes can itself be the implementation of another parameterized class (name it as A). For example:
  ```java
  class A<T> {
      B<T> x;
      ......
  }
  ``
  Note: A and B are different name.
  If so, class A should be parsed twice. Firstly, class A itself should be instantiated for other users of the class A. Secondly when instantiating, the paramters A obtained should be sent to class B so that it can be instantiated using these parameters.
- If class A and B are the same in 3), it will be a recursive definition of paramterized class A. In this case, instead of passing parameters of A to B, we should instantiate them at the same time.
  More complicated cases are not permitted. For example:
  ```java
  class A<T> {
      B<T> x;
      ......
  }
  ``
  ```java
  class B<S> {
      A<S> y;
      ......
  }
  ```
- If class A and B are the same in 3), it will be a recursive definition of parameterized class A. In this case, instead of passing parameters of A to B, we should instantiate them at the same time.

### 3.4.6 Ongoing work in special compiler

The following is part of the type checking related to ParTy that is to be done in special compiler:

- Checking for constraints when compiling a user program what use instances of parameterized class.
  1) “has” constraints, by looking at the method_info field of the class file related to the
type parameter.
   2) “extends” constraints, by looking at the super_class field of the class file related to
the type parameter, and of all the cascaded super classes.
   3) “implements” constraints, similar to the above, but also needs to look at the
interfaces field.

- Checking in Inheritance of Parameterized class:

   In the case of class \(<S>\) A extends class B\{…\}, no additional work is needed. In the case
of class \(<S>\) A extends class \(<T>\) B\{…\}, it is merely no use. The following case is usual
and useful:

   class A\(<T>\) extends B\(<T>\) \{…\}

   the type checking and instantiation of both class A and class B is needed by looking at
super_class field of class A, which is class B, and then instantiate class B consequently. If B
also extends a parameterized class, the cascaded type checking and instantiation are also
needed.

4 Evaluation

We evaluated ParTy in terms of its expressiveness and its performance. To measure the power of
expression added in a solution that provides only limited support for type parameterization, we
attempted to model several standard classes that are often defined generically in languages that
support genericity. To measure performance, we tested the execution speed of generic ParTy
code against Java code that attempts to provide the same flexibility through casting between the
actual type and type Object. Overall, we found that ParTy is a successful tool for designing
efficient, generic code, and that our technique improves execution speed by up to 27% over
standard Java.

4.1 Qualitative Evaluation

In testing the usefulness of ParTy code extensions, our group attempted to implement several
classes that are typically defined generically in other languages that support type
parameterization. We were able to easily create generic versions of the pair class, linked list
class, linked list iterator class, and the ordered set class, all in ParTy. The Test1 class is a simple
illustrative ParTy example that uses type parameterization.

/* Test1.pjava */
///////////////////////////////////////////////////////
// Test1.pjava is a generic list class that holds
// an array of type Type1. It provides a method to sort
// the members of the class. In additions to ParTy, Test1 would
// use the where clause to specify a constraint requiring a
// comparison operation from any type used to instantiate a
// Test1 object.

import java.lang.String;
class Test1<Type1> {
public Type1[] sort;
public Test1<Type1> (Type1[] x)
{
    int len=x.length;
    sort = new Type1[len];
    for(int i=0; i<len; i++)
    sort[i]=x[i];
}

void sort()
{
    Type1 temp;
    int count; // for checking whether interchanges are done or not
    int n; // for counting number of passes
    n=0;
    count=1;
    while (n < sort.length-1 || count != 0)
    {
        count = 0;
        for(int j=0; j<sort.length-1; j++)
        {
            if (sort[j].compareTo(sort[j+1])>0 )
            {
                temp = sort[j];
                sort[j] = sort[j+1];
                sort[j+1] = temp;
                count++;
            }
        }
        n++;
    }
}

/* TestC.pjava */


import java.lang.String;
import java.lang.Math;

class TestC {
    public static void main(String[] args) {
        int i;
        int temp;

        Int[] a = new Int[10];
        for(i=0; i<10; i++)
        {
            Int tmp =new Int();
            int j=(int)(Math.random()*100);
            tmp.Set_Val(j);
            a[i]=tmp;
        }
}
Test1<Int> y = new Test1<Int>(a);
System.out.println("For the Int Array");
System.out.println("Before Sorting:");
for(i=0; i<10; i++)
{
    temp=a[i].Get_Val();
    System.out.print(temp);
    System.out.print(" ");
}
System.out.println(" ");
y.sort();
System.out.println("After Sorting:");
for(i=0; i < 10; i++)
{
    temp=y.sort[i].Get_Val();
    System.out.print(temp);
    System.out.print(" ");
}
System.out.println(" ");
System.out.println("For the String Array");
String s[] = {"apple", "cell", "wonderful", "cool", "dare", "cable", "high", "TALL", "Jump");
System.out.println("Before Sorting:");
for (i=0; i<s.length; i++)
{
    System.out.print(s[i]);
    System.out.print(" ");
}
System.out.println(" ");
Test1<java0lang0String> z = new Test1<java0lang0String>(s);
z.sort();
System.out.println("After Sorting:");
for(i=0; i < z.sort.length; i++)
{
    System.out.print(z.sort[i]);
    System.out.print(" ");
}
System.out.println(" ");
/* Output of the User Program */
For the Int Array
Before Sorting:
84 60 59 60 32 55 2 85 61 25
After Sorting:
2 25 32 55 59 60 60 61 84 85

For the String Array
Before Sorting:
apple cell wonderful cool dare cable high TALL Jump
After Sorting:
Jump TALL apple cable cell cool high wonderful

Despite the success of ParTy in writing generic code, we found that some complicated code could not easily be written in ParTy. Because ParTy does not support where clause constraints for individual methods of a class, a user cannot write code that distinguishes different restrictions for different methods. The overall set of method constraints can be used in a class constraint declaration, but this is overly restrictive. Furthermore, a user cannot currently write code that instantiates a parameterized class using implicit type information; instead, all type information in ParTy must be explicit. We omitted these options after a careful review of genericity in other languages. These features, although powerful, were not deemed valuable because they would add unnecessary complexity to the original Java language.

4.2 Quantitative Evaluation

We performed a simple benchmark to examine the performance difference between generic ParTy code and the corresponding Java hack that programmers must use to simulate genericity in Java. Currently, type parameterization is simulated by creating classes that operate on Objects. Any type derived from Object can be passed to these classes at instantiation time, and even primitive types can be passed by using the corresponding wrapper class appropriately. However, the type casting overhead that the current Java method requires is high, and the benchmark reveals its cost, relative to the performance of ParTy code. Figure 6 illustrates the benchmark code in ParTy and Java, and figure 7 demonstrates the benchmarking results.

```java
public class cast {
    public static void main(String[] args) {
        Integer myInt = new Integer(0);
        int i;
        long st, ft;
        ListNodeWithCasting IntegerNode =
                new ListNodeWithCasting(myInt);
        st = System.currentTimeMillis();
        for( i=0; i<10000000; i++) {
            myInt = (Integer)IntegerNode.value;
        }
    }
}

public class nocast {
    public static void main(String[] args) {
        Integer myInt = new Integer(0);
        int i = 0;
        long st, ft;
        ListNode_P_java0lang0Integer ntegerNode =
                new ListNode_P_java0lang0Integer(myInt);
        st = System.currentTimeMillis();
        for( i=0; i<10000000; i++) {
            myInt = IntegerNode.value;
        }
    }
}
```
Figure 6. Benchmark code used in quantitative evaluation.

<table>
<thead>
<tr>
<th>Java: cast.java average execution speed</th>
<th>ParTy: nocast.pjava average execution speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>4527.8 milliseconds</td>
<td>3279.0 milliseconds</td>
</tr>
</tbody>
</table>

Figure 7. Benchmarking results.

From our results, we can conclude that ParTy yields up to 27.6% faster code than the corresponding Java code. It should be noted that this figure does not include the extra cost of running the ParTy load-time preprocessor. Because our preprocessor is experimental, the code has not been tuned for performance, and is not suitable for performance testing. Additionally, this figure does not include the one-time cost of running the special ParTy compiler. However, this cost does not effect run-time performance, and was omitted from the evaluation. However, there is reason to believe that even after including the extra cost of ParTy load-time instantiation, the general execution time of ParTy code will still remain lower than the corresponding Java code, although this must be pursued in further research.

5 Remaining Work

In order to create a robust system that will ultimately be accessible to all Java users, more work remains to be done. The current implementation does not provide an automated ParTy compiler. This is required for our language extension to be truly accessible to current Java users, and is currently in development at the time of this paper’s creation. Among other things, the compiler should enforce the type constraints specified by the where clause through added checks. Although the checks are defined semantically in section 3.3, they have not been implemented in the current working version of ParTy.

Furthermore, support for primitive type instantiation of a type parameter must be added. Although this is tedious work, it has already been proven to be possible [BLM97]. Due to time constraints, this feature was omitted from the current working version of ParTy.
Lastly, we would like to integrate the load-time preprocessor into the JVM ClassLoader. This will allow more benchmarking for comparing the overall speed of ParTy code running in the modified JVM to the original Java code running in the original JVM. Also, this would allow us to present ParTy as one complete package.

6 Conclusion

In Java, a programmer who wants to define a group of related classes that have similar behavior but manipulate different types cannot easily do so. In this paper, we extend Java with a mechanism to facilitate the construction of generic classes and interfaces. The paper provides a complete design of the language extension, including the formal syntax and semantics. We chose to add only the most useful extensions to Java, and we did not add complexity to the language unless its usefulness outweighs its complexity.

We also describe some details of our particular implementation. By using a load-time preprocessor, we are able to postpone code blowup until run time, and achieve observed preliminary performance increases as high as 27% over the corresponding Java code with type casting. Most importantly, the load-time processing avoids the need to add, change, or remove any Java bytecodes, making our ParTy implementation backwards and forwards compatible with existing Java code.

We found that the constructs that ParTy provides facilitates the programming of the types of classes most users want to make generic. We provide one simple example, and demonstrate that our language supports the tools necessary to build other generic classes, such as a linked list.

Finally, we discuss the work that remains in ParTy to provide a fully robust language extension. We believe that ParTy provides a useful, simple, and efficient method of writing generic code in Java, and hope that our work will make Java a better language.

Acknowledgements

We would like to thank the researchers at Sun Microsystems, Stanford University, Massachusetts Institute of Technology, Lucent Technology, and University of South Australia for pioneering work in Java genericity, as well as David Evans, for directing our work. We would also like to thank James McCliggot for constructing a pleasing name for our research project.

References


[BOSW98a] Brasha, Gilad, Odersky, Martin, Stoutamire, David, and Wadler, Philip. GJ: Extending the Java programming language with type parameters. Available at the GJ web site.


Song Li, Ying Lu, Hexin Wang, Michael Walker

University of Virginia
CS 655: Programming Languages